# Exim's interface to mail filtering

Exim is a mail transfer agent for Unix systems. This document describes the user interface to its in-built mail filtering facility, and is copyright © University of Cambridge 2001. It corresponds to Exim version 3.30.

---

## 1. Introduction

Most Unix mail transfer agents (programs that deliver mail) permit individual users to specify automatic forwarding of their mail, usually by placing a list of forwarding addresses in a file called **.forward** in their home directories. Exim extends this facility by allowing the forwarding instructions to be a set of rules rather than just a list of addresses, in effect providing '**.forward** with conditions'. Operating the set of rules is called *filtering*, and the file that contains them is called a *filter file*.

The ability to use filtering has to be enabled by the system administrator, and some of the individual facilities can be separately enabled or disabled. A local document should be provided to describe exactly what has been enabled. In the absence of this, consult your system administrator.

It is important to realize that no deliveries are actually made while a filter file is being processed. The result of filtering is a list of destinations to which a message should be delivered – the deliveries themselves take place later, along with all other deliveries for the message. This means that it is not possible to test for successful deliveries while filtering. It also means that duplicate addresses generated by filtering are dropped, as with any other duplicate addresses.

This document describes how to use a filter file and the format of its contents. It is intended for use by end-users. How the system administrator can set up and control the use of filtering is described in the full Exim specification.

## 2. Testing a new filter file

Filter files, especially the more complicated ones, should always be tested, as it is easy to make mistakes. Exim provides a facility for preliminary testing of a filter file before installing it. This tests the syntax of the file and its basic operation, and can also be used with ordinary **.forward** files.

Because a filter can do tests on the content of messages, a test message is required. Suppose you have a new filter file called **new-filter** and a test message called **test-message**. Assuming that Exim is installed with the conventional path name **/usr/lib/sendmail** (some operating systems use **/usr/sbin/sendmail**), the following command can be used:

```
/usr/lib/sendmail -bf new-filter <test-message
```

The **-bf** option tells Exim that the following item on the command line is the name of a filter file which is to be tested. There is also a **-bF** option, which is similar, but which is used for testing system filter files, as opposed to user filter files, and which is therefore of use only to the system administrator.

The test message is supplied on the standard input. If there are no message-dependent tests in the filter, then an empty file can be used. A supplied message must start with header lines or the 'From' message separator line which is found in many multi-message folder files. Note that blank lines at the start terminate the header lines. A warning is given if no headers are read.

The result of running this command, provided no errors are detected in the filter file, is a list of the actions that Exim would try to take if presented with the message for real. For example, the output

```
Deliver message to: gulliver@lilliput.fict.example
Save message to: /home/lemuel/mail/archive
```

means that one copy of the message would be sent to **gulliver@lilliput.fict.example**, and another would be added to the file **/home/lemuel/mail/archive**, if all went well.

The actions themselves are not attempted while testing a filter file in this way; there is no check, for example, that any forwarding addresses are valid. If you want to know why a particular action is being taken, add the **-v** option to the command. This causes Exim to output the results of any conditional tests and to indent its output according to the depth if nesting of **if** commands. Further additional output from a filter test can be generated by the **testprint** command, which is described below.

When Exim is outputting a list of the actions it would take, if any text strings are included in the output, non-printing characters therein are converted to escape sequences. In particular, if any text string contains a newline character, this is shown as '\n' in the testing output.

When testing a filter in this way, Exim makes up an 'envelope' for the message. The recipient is by default the user running the command, and so is the sender, but the command can be run with the **-f** option to supply a different sender. For example,

```
/usr/lib/sendmail -bf new-filter -f islington@neverwhere <test-message
```

Alternatively, if the **-f** option is not used, but the first line of the supplied message is a 'From' separator from a message folder file (not the same thing as a `From:` header line), the sender is taken from there. If **-f** is present, the contents of any 'From' line are ignored.

The 'return path' is the same as the envelope sender, unless the message contains a `Return-path:` header, in which case it is taken from there. You need not worry about any of this unless you want to test out features of a filter file that rely on the sender address or the return path.

It is possible to change the envelope recipient by specifying further options. The **-bfd** option changes the domain of the recipient address, while the **-bfl** option changes the 'local part', that is, the part before the @ sign. An adviser could make use of these to test someone else's filter file.

The **-bfp** and **-bfs** options specify the prefix or suffix for the local part. These are relevant only when support for multiple personal mailboxes is implemented; see the description in section 26 below.

### 3. Installing a filter file

A filter file is normally installed under the name **.forward** in your home directory – it is distinguished from a conventional **.forward** file by its first line (described below). However, the file name is configurable, and some system administrators may choose to use some different name or location for filter files.

### 4. Testing an installed filter file

Testing a filter file before installation cannot find every potential problem; for example, it does not actually run commands to which messages are piped. Some 'live' tests should therefore also be done once a filter is installed.

If at all possible, test your filter file by sending messages from some *other* account. If you send a message to yourself from the filtered account, and delivery fails, the error message will be sent back to the same account, which may cause another delivery failure. It won't cause an infinite sequence of such messages, because delivery failure messages do not themselves generate further messages. However, it does mean that the failure won't be returned to you, and also that the postmaster will have to investigate the stuck message.

If you have to test a filter from the same account, then a sensible precaution is to include the line

```
if error_message then finish endif
```

as the first filter command, at least while testing. This causes filtering to be abandoned for a delivery failure message, and since no destinations are generated, the message goes on to get delivered to the original address. Unless there is a good reason for not doing so, it is recommended that the above test be left in all filter files.

## 5. Format of filter files

Apart from leading white space, the first text in a filter file must be

```
# Exim filter
```

This is what distinguishes it from a conventional **.forward** file. If the file does not have this initial line it is treated as a conventional **.forward** file, both when delivering mail and when using the **-bf** testing mechanism. The white space in the line is optional, and any capitalization may be used. Further text on the same line is treated as a comment. For example, you could have

```
#   Exim filter   <<== do not edit or remove this line!
```

The remainder of the file is a sequence of filtering commands, which consist of keywords and data values, separated by white space or line breaks, except in the case of conditions for the `if` command, where round brackets (parentheses) also act as separators. For example, in the command

```
deliver gulliver@lilliput.fict.example
```

the keyword is `deliver` and the data value is `gulliver@lilliput.fict.example`. The commands are in free format, and there are no special terminators. If the character # follows a separator, then everything from # up to the next newline is ignored. This provides a way of including comments in a filter file.

There are two ways in which a data value can be input:

- If the text contains no white space then it can be typed verbatim. However, if it is part of a condition, it must also be free of round brackets (parentheses), as these are used for grouping in conditions.
- Otherwise it must be enclosed in double quotation marks. In this case, the character \ (backslash) is treated as an ‘escape character’ within the string, causing the following character or characters to be treated specially:

  \n is replaced by a newline
  \r is replaced by a carriage return
  \t is replaced by a tab

  Backslash followed by up to three octal digits is replaced by the character specified by those digits, and \x followed by up to two hexacimal digits is treated similarly. Backslash followed by any other character is replaced by the second character, so that in particular, \" becomes " and \\ becomes \. A data item enclosed in double quotes can be continued onto the next line by ending the first line with a backslash. Any leading white space at the start of the continuation line is ignored.

In addition to the escape character processing that occurs when strings are enclosed in quotes, most data values are also subject to *string expansion* (as described in the next section), in which case the characters **$** and \ are also significant. This means that if a single backslash is actually required in such a string, and the string is also quoted, \\\\ has to be entered.

## 6. String expansion

Most data values are expanded before use. Expansion consists of replacing substrings beginning with **$** with other text. The full expansion facilities are described from section 29 below onwards, but the most common case is the substitution of a simple variable. For example, the substring

```
$reply_address
```

is replaced by the address to which replies to the message should be sent. If such a variable name is followed by a letter or digit or underscore, it must be enclosed in curly brackets (braces), for example,

```
${reply_address}
```

If a $ character is actually required in an expanded string, it must be escaped with a backslash, and because backslash is also an escape character in quoted input strings, it must be doubled in that case. The following two examples illustrate the two cases:

```
if $local_part contains \$ then ...
if $local_part contains "\\$" then ...
```

The variable substitutions most likely to be useful in filter files are:

**$home**: The user's home directory.

**$local_part**: The part of the email address that precedes the @ sign – normally the user's login name. If support for multiple personal mailboxes is enabled (see section 26 below) and a prefix or suffix for the local part was recognized, it is removed from the string in this variable.

**$local_part_prefix**: If support for multiple personal mailboxes is enabled (see section 26 below), and a local part prefix was recognized, then this variable contains the prefix. Otherwise it contains an empty string.

**$local_part_suffix**: If support for multiple personal mailboxes is enabled (see section 26 below), and a local part suffix was recognized, then this variable contains the suffix. Otherwise it contains an empty string.

**$message_body**: The initial portion of the body of the message. By default, up to 500 characters are read into this variable, but the system administrator can configure this to some other value. Newlines in the body are converted into single spaces.

**$message_body_end**: The final portion of the body of the message, formatted and limited in the same way as **$message_body**.

**$message_body_size**: The size of the body of the message, in bytes.

**$message_headers**: The header lines of the message, concatenated into a single string, with newline characters between them.

**$message_id**: The message's local identification string, which is unique for each message handled by a single host.

**$message_size**: The size of the entire message, in bytes.

**$original_local_part**: When a top-level address is being processed, this contains the same value as the variable **$local_part**. However, if an address generated by an alias, forward, or filter file is being processed, this variable contains the local part of the original address.

**$reply_address**: The contents of the `Reply-to:` header, if the message has one; otherwise the contents of the `From:` header. It is the address to which normal replies to the message should be sent.

**$return_path**: The return path – that is, the sender field that will be sent as part of the message's envelope, and which is the address to which delivery errors are sent. In many cases, this has the same value as **$sender_address**, but if, for example, an incoming message to a mailing list has been expanded, then **$return_path** may contain the address of the list maintainer instead.

**$sender_address**: The sender address that was received in the envelope of the message. This is not necessarily the same as the contents of the `From:` or `Sender:` header lines. For delivery error messages ('bounce messages') there is no sender address, and this variable is empty.

**$tod_full**: A full version of the time and date, for example: Wed, 18 Oct 1995 09:51:40 +0100. The timezone is always given as a numerical offset from GMT.

**$tod_log**: The time and date in the format used for writing Exim's log files, for example: 1995-10-12 15:32:29.

### 7. Header variables

There is a special set of expansion variables containing the headers of the message being processed. These variables have names beginning with `$header_` followed by the name of the header, terminated by a colon. For example,

```
$header_from:
$header_subject:
```

The whole item, including the terminating colon, is replaced by the contents of the message header. If there is more than one header with the same name, their contents are concatenated, with a single newline character between them. The capitalization of the name following `$header_` is not significant. Because any printing character except colon may appear in the name of a message's header (this is a requirement of RFC 822, the document that describes the format of a mail message) curly brackets must *not* be used in this case, as they will be taken as part of the header name. Two shortcuts are allowed in naming header variables:

- The initiating `$header_` can be abbreviated to `$h_`.
- The terminating colon can be omitted if the next character is white space. The white space character is retained in the expanded string. However, this is not recommended, because it makes it easy to forget the colon when it really is needed.

If the message does not contain a header of the given name, an empty string is substituted. Thus it is important to spell the names of headers correctly. Do not use `$header_Reply_to` when you really mean `$header_Reply-to`.

You can test for the presence or absence of a header by means of the 'def' condition, which is described in section 32.

### 8. User variables

There are ten user variables with names **$n0** – **$n9** that can be incremented by the **add** command (see section 11). These can be used for 'scoring' messages in various ways. If Exim is configured to run a 'system filter' on every message, the values left in these variables are copied into the variables **$sn0** – **$sn9** at the end of the system filter, thus making them available to users' filter files. How these values are used is entirely up to the individual installation.

### 9. Significant deliveries

When in the course of delivery a message is processed by a filter file, what happens next, that is, after the whole filter file has been processed, depends on whether the filter has set up any *significant deliveries* or not. If there is at least one significant delivery, then the filter is considered to have handled the entire delivery arrangements for the current address, and no further processing of the address takes place. If, however, no significant deliveries have been set up, Exim continues processing the current address as if there were no filter file, and typically sets up a delivery of a copy of the message into a local mailbox. In particular, this happens in the special case of a filter file containing only comments.

The delivery commands **deliver**, **save**, and **pipe** are by default significant. However, if such a command is preceded by the word `unseen`, then its delivery is not considered to be significant. In contrast, other commands such as `mail` and `vacation` do not count as significant deliveries unless preceded by the word `seen`.

### 10. Filter commands

The filter commands which are described in subsequent sections are listed below, with the section in which they are described in brackets:

| | |
|---|---|
| **add** | increment a user variable (11) |
| **deliver** | deliver to an email address (12) |
| **finish** | end processing (17) |
| **if** | test condition(s) (18) |
| **logfile** | define log file (16) |
| **logwrite** | write to log file (16) |
| **mail** | send a reply message (15) |
| **pipe** | pipe to a command (14) |
| **save** | save to a file (13) |
| **testprint** | print while testing (17) |
| **vacation** | tailored form of **mail** (15) |

In addition, when Exim's filtering facilities are being used as a system filter, the **fail**, **freeze**, and **headers** commands are available. However, since they are usable only by the system administrator and not by ordinary users, they are described in the main Exim specification rather than in this document.

### 11. The add command

```
      add <number> to <user variable>
e.g. add 2 to n3
```

There are 10 user variables of this type, and their values can be obtained by the normal expansion syntax (for example **$n3**) in other commands. At the start of filtering, these variables all contain zero. Both arguments of the **add** command are expanded before use, making it possible to add variables to each other. Subtraction can be obtained by adding negative numbers.

### 12. The deliver command

```
      deliver <mail address>
e.g. deliver "Dr Livingstone <David@somewhere.africa>"
```

This provides a forwarding operation. The message is sent on to the given address, exactly as happens if the address had appeared in a traditional **.forward** file. To deliver a copy of the message to your normal mailbox, your login name can be given. Once an address has been processed by the filtering mechanism, an identical generated address will not be so processed again, so doing this does not cause a loop.

However, if you have a mail alias, you should *not* refer to it here. For example, if the mail address `L.Gulliver` is aliased to `lg103` then all references in Gulliver's **.forward** file should be to `lg103`. A reference to the alias will not work for messages that are addressed to that alias, since, like **.forward** file processing, aliasing is performed only once on an address, in order to avoid looping.

Only a single address may be given to a **deliver** command, but multiple occurrences of the command may be used to cause the message to be delivered to more than one address. However, duplicate addresses are discarded.

Following the new address, an optional second address, preceded by `errors_to` may appear. This changes the address to which delivery errors on the forwarded message will be sent. Instead of going to the message's original sender, they go to this new address. For ordinary users, the only value that is permitted for this address is the user whose filter file is being processed. For example, the user `lg103` whose mailbox is in the domain **lilliput.example** could have a filter file that contains

```
deliver jon@elsewhere.example errors_to lg103@lilliput.example
```

Clearly, using this feature makes sense only in situations where some (but not all) messages are being forwarded. In particular, bounce messages must not be forwarded in this way, as this is likely to create a mail loop if something goes wrong.

**13. The save command**

```
      save <file name>
e.g. save $home/mail/bookfolder
```

This causes a copy of the message to be appended to the given file (that is, the file is used as a mail folder). More than one **save** command may appear; each one causes a copy of the message to be written to its argument file, provided they are different (duplicate **save** commands are ignored).

If the file name does not start with a / character, then the contents of the **$home** variable are prepended. The user must of course have permission to write to the file, and the writing of the file takes place in a process that is running as the user, under the user's primary group. Any secondary groups to which the user may belong are not normally taken into account, though the system administrator can configure Exim to set them up. In addition, the ability to use this command at all is controlled by the system administrator – it may be forbidden on some systems. An optional mode value may be given after the file name, for example,

```
save /some/folder 0640
```

The value for the mode is interpreted as an octal number, even if it does not begin with a zero. This makes it possible for users to override the system-wide mode setting for file deliveries, which is normally 600. If an existing file does not have the correct mode, it is changed.

An alternative form of delivery may be enabled on your system, in which each message is delivered into a new file in a given directory. If this is the case, this functionality can be requested by giving the directory name terminated by a slash after the **save** command, for example

```
save separated/messages/
```

There are several different formats for such deliveries; check with your system administrator or local documentation to find out which (if any) are available on your system. If this functionality is not enabled, the use of a path name ending in a slash causes an error.

**14. The pipe command**

```
      pipe <command>
e.g. pipe "$home/bin/countmail $sender_address"
```

This command causes a separate process to be run, and a copy of the message is passed on its standard input. The process runs as the user, under the user's primary group. Any secondary groups to which the user may belong are not normally taken into account, though the system administrator can configure Exim to set them up. More than one **pipe** command may appear; each one causes a copy of the message to be written to its argument pipe, provided they are different (duplicate **pipe** commands are ignored).

The command supplied to **pipe** is split up by Exim into a command name and a number of arguments, delimited by white space except for arguments enclosed in double quotes, in which case backslash is interpreted as an escape, or in single quotes, in which case no escaping is recognized. Note that as the whole command is normally supplied in double quotes, a second level of quoting is required for internal double quotes. For example:

```
pipe "$home/myscript \"size is $message_size\""
```

String expansion is performed on the separate components after the line has been split up, and the command is then run directly by Exim; it is not run under a shell. Therefore, substitution cannot change the number of arguments, nor can quotes, backslashes or other shell metacharacters in variables cause confusion.

Documentation for some programs that are normally run via this kind of pipe often suggest the the command start with

```
IFS=" "
```

This is a shell command, and should *not* be present in Exim filter files, since it does not normally run the command under a shell.

However, there is an option that the administrator can set to cause a shell to be used. In this case, the entire command is expanded as a single string and passed to the shell for interpretation. It is recommended that this be avoided if at all possible, since it can lead to problems when inserted variables contain shell metacharacters.

The default PATH set up for the command is determined by the system administrator, usually containing at least **/usr/bin** so that common commands are available without having to specify an absolute file name. However, it is possible for the system administrator to restrict the pipe facility so that the command name must not contain any / characters, and must be found in one of the directories in the configured PATH. It is also possible for the system administrator to lock out the use of the **pipe** command altogether.

When the command is run, the following environment variables are set up:

| | |
|---|---|
| DOMAIN | the local domain of the address |
| HOME | your home directory |
| LOCAL_PART | your login name |
| LOGNAME | your login name |
| MESSAGE_ID | the message's unique id |
| PATH | the command search path |
| SENDER | the sender of the message |
| SHELL | **/bin/sh** |
| USER | your login name |

If you run a command that is a shell script, be very careful in your use of data from the incoming message in the commands in your script. RFC 822 is very generous in the characters that are legally permitted to appear in mail addresses, and in particular, an address may begin with a vertical bar or a slash. For this reason you should always use quotes round any arguments that involve data from the message, like this:

```
/some/command "$SENDER"
```

so that inserted shell meta-characters do not cause unwanted effects.

The pipe command should return a zero completion code if all has gone well. Most non-zero codes are treated by Exim as indicating a failure of the pipe. This is treated as a delivery failure, causing the message to be returned to its sender. However, there are some completion codes which are treated as temporary errors. The message remains on Exim's spool disc, and the delivery is tried again later, though it will ultimately time out if the delivery failures go on too long. The completion codes to which this applies can be specified by the system administrator; the default values are 73 and 75.

The pipe command should not normally write anything to its standard output or standard error file descriptors. If it does, whatever is written is normally returned to the sender of the message as a delivery error, though this action can be varied by the system administrator.

### 15. Mail commands

There are two commands which cause the creation of a new mail message, neither of which count as a significant delivery unless the command is preceded by the word `seen`. This is a powerful facility, but it should be used with care, because of the danger of creating infinite sequences of messages. The system administrator can forbid the use of these commands altogether.

To help prevent runaway message sequences, these commands have no effect when the incoming message is a delivery error message, and messages sent by this means are treated as if they were reporting delivery errors. Thus they should never themselves cause a delivery error message to be returned. The basic mail-sending command is

```
mail [to <address-list>]
     [cc <address-list>]
     [bcc <address-list>]
     [from <address>]
     [reply_to <address>]
     [subject <text>]
     [text <text>]
     [[expand] file <filename>]
     [return message]
     [log <log file name>]
     [once <note file name>]
     [once_repeat <time interval>]
```

```
e.g. mail text "Your message about $h_subject has been received"
```

As a convenience for use in one common case, there is also a command called **vacation**. It behaves in the same way as **mail**, except that the defaults for the `file`, `log`, `once`, and `once_repeat` options are

```
expand file .vacation.msg
log  .vacation.log
once .vacation
once_repeat 7d
```

respectively. These are the same file names and repeat period used by the traditional Unix **vacation** command. The defaults can be overridden by explicit settings, but if a file name is given its contents are expanded only if explicitly requested. The **vacation** command is normally used conditionally, subject to the **personal** condition (see section 21 below) so as not to send automatic replies to non-personal messages from mailing lists or elsewhere.

For both commands, the key/value argument pairs can appear in any order. At least one of `text` or `file` must appear (except with `vacation`); if both are present, the text string appears first in the message. If `expand` precedes `file`, then each line of the file is subject to string expansion as it is included in the message.

Several lines of text can be supplied to `text` by including the escape sequence '\n' in the string where newlines are required. If the command is output during filter file testing, newlines in the text are shown as '\n'.

Note that the keyword for creating a `Reply-To:` header is **reply_to**, because Exim keywords may contain underscores, but not hyphens. If the **from** keyword is present and the given address does not match the user who owns the forward file, Exim normally adds a `Sender:` header to the message, though it can be configured not to do this.

If no `to` argument appears, the message is sent to the address in the `$reply_address` variable (see section 6 above). An `In-Reply-To:` header is automatically included in the created message, giving a reference to the message identification of the incoming message.

If **return message** is specified, the incoming message that caused the filter file to be run is added to the end of the message, subject to a maximum size limitation.

If a log file is specified, a line is added to it for each message sent.

If a `once` file is specified, it is used to hold a database for remembering who has received a message, and no more than one message is ever sent to any particular address, unless `once_repeat` is set. This specifies a time interval after which another copy of the message is sent. The interval is specified as a sequence of numbers, each followed by the initial letter of one of 'seconds', 'minutes', 'hours', 'days', or 'weeks'. For example,

```
once_repeat 5d4h
```

causes a new message to be sent if 5 days and 4 hours have elapsed since the last one was sent. There must be no white space in a time interval.

Commonly, the file name specified for `once` is used as the base name for direct-access (DBM) file operations. There are a number of different DBM libraries in existence. Some operating systems provide one as a default, but even in this case a different one may have been used when building Exim. With some DBM libraries, specifying `once` results in two files being created, with the suffixes `.dir` and `.pag` being added to the given name. With some others a single file with the suffix `.db` is used, or the name is used unchanged.

Using a DBM file for implementing the `once` feature means that the file grows as large as necessary. This is not usually a problem, but some system administrators want to put a limit on it. The facility can be configured not to use a DBM file, but instead, to use a regular file with a maximum size. The data in such a file is searched sequentially, and if the file fills up, the oldest entry is deleted to make way for a new one. This means that some correspondents may receive a second copy of the message after an unpredictable interval. Consult your local information to see if your system is configured this way.

More than one **mail** or **vacation** command may be obeyed in a single filter run; they are all honoured, even when they are to the same recipient.

### 16. Logging commands

A log can be kept of actions taken by a filter file. This facility is normally available in conventional configurations, but there are some situations where it might not be. Also, the system administrator may choose to disable it. Check your local information if in doubt.

Logging takes place while the filter file is being interpreted. It does not queue up for later like the delivery commands. The reason for this is so that a log file need be opened only once for several write operations. There are two commands, neither of which constitutes a significant delivery. The first defines a file to which logging output is subsequently written:

```
      logfile <file name>
e.g.  logfile $home/filter.log
```

The file name may optionally be followed by a mode for the file, which is used if the file has to be created. For example,

```
      logfile $home/filter.log 0644
```

The number is interpreted as octal, even if it does not begin with a zero. The default for the mode is 600. It is suggested that the **logfile** command normally appear as the first command in a filter file. Once **logfile** has been obeyed, the **logwrite** command can be used to write to the log file:

```
      logwrite "<some text string>"
e.g.  logwrite "$tod_log $message_id processed"
```

It is possible to have more than one **logfile** command, to specify writing to different log files in different circumstances. Writing takes place at the end of the file, and a newline character is added to the end of each string if there isn't one already there. Newlines can be put in the middle of the string by using the '\n' escape sequence. Lines from simultaneous deliveries may get interleaved in the file, as there is no interlocking, so you should plan your logging with this in mind. However, data should not get lost.

In earlier versions of Exim the **logwrite** command was called **log**, and this name remains available for backwards compatibility. However, it is not possible to use the name **log** as a command name following a **mail** command, because it will be interpreted as the **log** option of that command.

### 17. Other commands

The command `finish`, which has no arguments, causes Exim to stop interpreting the filter file. This is not a significant action unless preceded by `seen`. A filter file containing only `seen finish` is a black hole.

It is sometimes helpful to be able to print out the values of variables when testing filter files. The command

```
      testprint <text>
e.g. testprint "home=$home reply_address=$reply_address"
```

does nothing when mail is being delivered. However, when the filtering code is being tested by means of the **-bf** option, the value of the string is written to the standard output.

When Exim's filtering facilities are being used as a system filter, the **fail** and **freeze** commands are available. However, since they are usable only by the system administrator and not by ordinary users, they are described in the main Exim specification rather than in this document.

### 18. Obeying commands conditionally

Most of the power of filtering comes from the ability to test conditions and obey different commands depending on the outcome. The `if` command is used to specify conditional execution, and its general form is

```
if    <condition>
then  <commands>
elif  <condition>
then  <commands>
else  <commands>
endif
```

There may be any number of `elif` and `then` sections (including none) and the `else` section is also optional. Any number of commands, including nested `if` commands, may appear in any of the <*commands*> sections.

Conditions can be combined by using the words `and` and `or`, and round brackets (parentheses) can be used to specify how several conditions are to combine. Without brackets, `and` is more binding than `or`. For example,

```
if
  $h_subject: contains "Make money" or
  $h_precedence: is "junk" or
  ($h_sender: matches ^\\d{8}@ and not personal) or
  $message_body contains "this is spam"
then
  seen finish
endif
```

A condition can be preceded by `not` to negate it, and there are also some negative forms of condition that are more English-like.

### 19. String testing conditions

There are a number of conditions that operate on text strings, using the words 'begins', 'ends', 'is', 'contains' and 'matches'. If the condition names are written in lower-case, the testing of letters is done without regard to case; if they are written in upper-case (for example, 'CONTAINS') then the case of letters is significant.

```
      <text1> begins <text2>
      <text1> does not begin <text2>
e.g. $header_from: begins "Friend@"
```

A 'begins' test checks for the presence of the second string at the start of the first, both strings having been expanded.

```
      <text1> ends <text2>
      <text1> does not end <text2>
  e.g. $header_from: ends "public.com"
```

An ‘ends’ test checks for the presence of the second string at the end of the first, both strings having been expanded.

```
      <text1> is <text2>
      <text1> is not <text2>
  e.g. $local_part_suffix is "-foo"
```

An ‘is’ test does an exact match between the strings, having first expanded both strings.

```
      <text1> contains <text2>
      <text1> does not contain <text2>
  e.g. $header_subject: contains "evolution"
```

A ‘contains’ test does a partial string match, having expanded both strings.

```
      <text1> matches <text2>
      <text2> does not match <text2>
  e.g. $sender_address matches "(Bill|John)@"
```

For a ‘matches’ test, after expansion of both strings, the second one is interpreted as a regular expression. Exim uses the PCRE regular expression library, which provides regular expressions that are compatible with Perl.

Care must be taken if you need a backslash in a regular expression, because backslashes are interpreted as escape characters both by the string expansion code and by Exim’s normal string reading code. For example, if you want to test the sender address for a domain ending in `.com` the regular expression is

```
  \.com$
```

The backslash and dollar sign in that expression have to be escaped when used in a filter command, as otherwise they would be interpreted by the expansion code. Thus what you actually write is

```
  if $sender_address matches \\.com\$
```

However, if the expression is given in quotes (mandatory only if it contains white space) you have to write

```
  if $sender_address matches "\\\\.com\\$"
```

with ‘\\\\’ for a backslash and ‘\\$’ for a dollar sign. Hence, if you actually require the string ‘\$’ in a regular expression that is given in double quotes, you need to write ‘\\\\\\$’.

If the regular expression contains bracketed sub-expressions, then numeric variable substitutions such as **$1** can be used in the subsequent actions after a successful match. If the match fails, the values of the numeric variables remain unchanged. Previous values are not restored after `endif` – in other words, only one set of values is ever available. If the condition contains several sub-conditions connected by `and` or `or`, it is the strings extracted from the last successful match that are available in subsequent actions. Numeric variables from any one sub-condition are also available for use in subsequent sub-conditions, since string expansion of a condition occurs just before it is tested.

**20. Numeric testing conditions**

The following conditions are available for performing numerical tests:

```
      <number1> is above <number2>
      <number1> is not above <number2>
      <number1> is below <number2>
      <number1> is not below <number2>
  e.g. $message_size is not above 10k
```

The <*number*> arguments must expand to strings of digits, optionally followed by one of the letters K or M (upper-case or lower-case) which cause multiplication by 1024 and 1024x1024 respectively.

### 21. Testing for personal mail

A common requirement is to distinguish between incoming personal mail and mail from a mailing list. In particular, this test is normally required for so-called ‘vacation messages’. The condition

```
personal
```

is a shorthand for

```
$header_to: contains $local_part@$domain and
$header_from: does not contain $local_part@$domain and
$header_from: does not contain server@ and
$header_from: does not contain daemon@ and
$header_from: does not contain root@ and
$header_subject: does not contain "circular" and
$header_precedence: does not contain "bulk" and
$header_precedence: does not contain "list" and
$header_precedence: does not contain "junk"
```

The variable `$local_part` contains the local part of the mail address of the user whose filter file is being run – it is normally your login id. The `$domain` variable contains the mail domain. This condition tests for the appearance of the current user in the `To:` header, checks that the sender is not the current user or one of a number of common daemons, and checks the content of the `Subject:` and `Precedence:` headers.

If prefixes or suffixes are in use for local parts – something which depends on the configuration of Exim (see section 26 below) – then the first two tests above are also done with

```
$local_part_prefix$local_part$local_part_suffix
```

instead of just **$local_part**. If the system is configured to rewrite local parts of mail addresses, for example, to rewrite ‘dag46’ as ‘Dirk.Gently’, then the rewritten form of the address is also used in the tests.

This example shows the use of **personal** in a filter file that is sending out vacation messages:

```
if personal then
  mail
    to $reply_address
    subject "Re: $h_subject:"
    file $home/vacation/message
    once $home/vacation/once
    once_repeat 10d
endif
```

It is quite common for people who have mail accounts on a number of different systems to forward all their mail to one system, and in this case a check for personal mail should test all their various mail addresses. To allow for this, the **personal** condition keyword can be followed by

```
alias <address>
```

any number of times, for example

```
personal alias smith@else.where alias jones@other.place
```

This causes messages containing the alias addresses in any places where the local address is tested to be treated as personal.

### 22. Testing for significant deliveries

Whether or not any previously obeyed filter commands have resulted in a significant delivery can be tested by the condition `delivered`, for example:

```
if not delivered then save mail/anomalous endif
```

### 23. Testing for error messages

The condition `error_message` is true if the incoming message is a mail delivery error message. Putting the command

```
if error_message then finish endif
```

at the head of your filter file is a useful insurance against things going wrong in such a way that you cannot receive delivery error reports, and is highly recommended. Note that **error_message** is a condition, not an expansion variable, and therefore is not preceded by **$**.

### 24. Testing delivery status

There are two conditions which are intended mainly for use in system filter files, but which are available in users' filter files as well. The condition `first_delivery` is true if this is the first attempt to deliver the message, and false otherwise. In a user filter file it will be false only if there was previously an error in the filter, or if a delivery for the user failed owing to, for example, a quota error, or forwarding to a remote address that was deferred for some reason.

The condition `manually_thawed` is true only if the message was 'frozen' for some reason, and was subsequently released by the system administrator. It is unlikely to be of use in users' filter files.

### 25. Testing a list of addresses

There is a facility for looping through a list of addresses and applying a condition to each of them. It takes the form

`foranyaddress` <*string*> (<*condition*>)

where <*string*> is interpreted as a list of RFC 822 addresses, as in a typical header line, and <*condition*> is any valid filter condition or combination of conditions. The parentheses surrounding the condition are mandatory, to delimit it from possible further sub-conditions of the enclosing **if** command. Within the condition, the expansion variable **$thisaddress** is set to the non-comment portion of each of the addresses in the string in turn. For example, if the string is

```
B.Simpson <bart@springfield>, lisa@springfield (his sister)
```

then **$thisaddress** would take on the values `bart@springfield` and `lisa@springfield` in turn.

If there are no valid addresses in the list, the whole condition is false. If the internal condition is true for any one address, the overall condition is true and the loop ends. If the internal condition is false for all addresses in the list, the overall condition is false. This example tests for the presence of an eight-digit local part in any address in a **To:** header:

```
if foranyaddress $h_to: ( $thisaddress matches ^\\d{8}@ ) then ...
```

When the overall condition is true, the value of **$thisaddress** in the commands that follow **then** is the last value it took on inside the loop. At the end of the **if** command, its value is reset to what it was before. It is best to avoid the use of multiple occurrences of **foranyaddress**, nested or otherwise, in a single **if** command, if the value of **$thisaddress** is to be used afterwards, because it isn't always clear what the value will be. Nested **if** commands should be used instead.

Header lines can be joined together if a check is to be applied to more than one of them. For example:

```
if foranyaddress $h_to:,$h_cc: ....
```

scans through the addresses in both the **To:** and the **Cc:** headers.

### 26. Multiple personal mailboxes

The system administrator can configure Exim so that users can set up variants on their email addresses and handle them separately. Consult your system administrator or local documentation to see if this facility is enabled on your system, and if so, what the details are.

The facility involves the use of a prefix or a suffix on an email address. For example, all mail addressed to **lg103-**<*something*> would be the property of user **lg103**, who could determine how it was to be handled, depending on the value of <*something*>.

There are two possible ways in which this can be set up. The first possibility is the use of multiple **.forward** files. In this case, mail to **lg103-foo**, for example, is handled by looking for a file called **.forward-foo** in **lg103's** home directory. If such a file does not exist, delivery fails and the message is returned to its sender.

The alternative approach is to pass all messages through a single **.forward** file, which must be a filter file in order to distinguish between the different cases by referencing the variables **$local_part_prefix** or **$local_part_suffix**, as in the final example in section 28 below. If the filter file does not handle a prefixed or suffixed address, delivery fails and the message is returned to its sender.

It is possible to configure Exim to support both schemes at once. In this case, a specific **.forward-foo** file is first sought; if it is not found, the basic **.forward** file is used.

The `personal` test (see section 21) includes prefixes and suffixes in its checking.

### 27. Ignoring delivery errors

As was explained above, filtering just sets up addresses for delivery – no deliveries are actually done while a filter file is active. If any of the generated addresses subsequently suffers a delivery failure, an error message is generated in the normal way. However, if the filter command which sets up a delivery is preceded by the word `noerror`, then errors for that delivery, *and any deliveries consequent on it* (that is, from alias, forwarding, or filter files it invokes) are ignored.

### 28. Examples of filter commands

Simple forwarding:

```
# Exim filter
deliver baggins@rivendell.middle.earth
```

Vacation handling using traditional means, assuming that the **.vacation.msg** and other files have been set up in your home directory:

```
# Exim filter
unseen pipe "/usr/ucb/vacation \"$local_part\""
```

Vacation handling inside Exim, having first created a file called **.vacation.msg** in your home directory:

```
# Exim filter
if personal then vacation endif
```

File some messages by subject:

```
# Exim filter
if $header_subject: contains "empire" or
   $header_subject: contains "foundation"
then
   save $home/mail/f&e
endif
```

Save all non-urgent messages by weekday:

```
# Exim filter
if $header_subject: does not contain "urgent" and
   $tod_full matches "^(...),"
then
  save $home/mail/$1
endif
```

Throw away all mail from one site, except from postmaster:

```
# Exim filter
if $reply_address contains "@spam.site" and
   $reply_address does not contain "postmaster@"
then
   seen finish
endif
```

Handle multiple personal mailboxes

```
# Exim filter
if $local_part_suffix is "-foo"
then
  save $home/mail/foo
elif $local_part_suffix is "-bar"
then
  save $home/mail/bar
endif
```

### 29. More about string expansion

The description which follows in the next section is an excerpt from the full specification of Exim, except that it lists only those expansion variables that are likely to be useful in filter files.

Expanded strings are 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 which is interpreted and replaced as described below.

An uninterpreted dollar can be included in the string by putting a backslash in front of it – if the string appears in quotes, two backslashes are required because the quotes themselves cause interpretation of backslashes when the string is read in. A backslash can be used to prevent any special character being treated specially in an expansion, including itself.

A backslash followed by one of the letters ‘n’, ‘r’, or ‘t’ 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, while a backslash followed by ‘x’ and up to two hexadecimal digits is a hexadecimal encoding. A backslash followed by any other character causes that character to be added to the output string uninterpreted. These escape sequences are also recognized in quoted strings as they are read in; their interpretation in expansions as well is useful for unquoted strings and other cases such as looked-up strings that are then expanded.

### 30. 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. Within braces, however, 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 curly brackets) is available only for variables; it does *not* apply to message headers. The names of the variables are given in section 33 below. If the name of a non-existent variable is given, the expansion fails.

**$header_**<*header name*>**:** or **$h_**<*header name*>**:**

Substitute the contents of the named message header line, for example

```
$header_reply-to:
```

The header names follow the syntax of RFC 822, 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.

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 32 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, with a newline character between each of them. However, if the length of this string exceeds 64K, any further headers of the same name are ignored.

**${**<*op*>**:**<*string*>**}**

The string is first itself expanded, and then the operation specified by <*op*> is applied to it. For example,

```
${lc:$local_part}
```

A list of operators is given in section 31 below. The string starts with the first character after the colon, which may be leading white space.

**${extract{**<*key*>**}{**<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**

The key and <*string1*> are first expanded separately. The key must not consist entirely of digits. For the string, the result must be of the form:

```
<key1> = <value1>  <key2> = <value2> ...
```

where the equals signs and spaces 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 5. The expanded <*string1*> is searched for the value that corresponds to the key. If it 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 looked up 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 }
```

{<*string2*>} must be present for ‘fail’ to be recognized. When this syntax is used, if the extraction fails, the entire expansion fails. This causes processing of the filter file to fail, which either delays delivery of the message, or causes the filter file to be ignored, depending on how the administrator has configured Exim.

**${extract{**<*number*>**}{**<*separators*>**}{**<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**

The <*number*> argument must consist entirely of decimal digits. 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. The first field is numbered one. If 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; if it is zero, the entire string is returned. The fields in the string are separated by any one of the characters in the separator string. For example:

```
${extract{2}{:}{x:42:99:& Mailer::/bin/bash}}
```

yields '42'. Two successive separators mean that the field between them is empty (for example, the fifth field above).

**${if** <*condition*> **{**<*string1*>**}{**<*string2*>**}}**

If <*condition*> is true, <*string1*> is expanded and replaces the whole item; otherwise <*string2*> is used. 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. The available conditions are described in section 32 below.

**${lookup{**<*key*>**}** <*search type*> **{**<*file*>**} {**<*string1*>**} {**<*string2*>**}}**

**${lookup** <*search type*> **{**<*query*>**} {**<*string1*>**} {**<*string2*>**}}**

These items specify data lookups in files and databases, as discussed in chapter 6 of the main Exim specification. 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, 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 null on failure. Alternatively, <*string2*> 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. 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 6.6 and 6.7 of the main Exim specification).

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}"
```

**${lookup{**<*key:subkey*>**}** <*search type*> **{**<*file*>**} {**<*string1*>**} {**<*string2*>**}}**

This is just a syntactic variation for a single-key lookup, surrounded by an **extract** item. It searches for <*key*> in the file as described above for single-key lookups; if it succeeds, it extracts from the data a subfield which is identified by the <*subkey*>. For example, if a line in a linearly searched file contains

```
alice: uid=1984 gid=2001
```

then expanding the string

```
${lookup{alice:uid}lsearch{<file name>}{$value}}
```

yields the string '1984'. If the subkey is not found in the looked up data, then <*string2*>, if present, is expanded and replaces the entire item. Otherwise the replacement is null. The example above could equally well be written like this:

```
${extract{uid}{${lookup{alice}lsearch{<file name>}}}}
```

and this is recommended, because this approach can also be used with query-style lookups.

**${sg{**<*subject*>**}{**<*regex*>**}{**<*replacement*>**}}**

This item works like Perl's substitution operator (s) with the global (/g) option; hence its name. It 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}{(\\d+)=}{K\$1=}}
```

yields 'K1=A K4=D K3=C'.

**${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.

### 31. Expansion operators

The following operations can be performed on portions of an expanded string. The substring is first expanded before the operation is applied to it.

**${domain:**<*string*>**}**

The string is interpreted as an RFC 822 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.

**${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.

**${hash_**<*n*>**_**<*m*>**:**<*string*>**}**

The two items <*n*> and <*m*> are numbers. If <*n*> is greater than or equal to the length of the string, the operator 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
```

and if <*m*> is not present the value 26 is used, so that only lower case letters appear. These examples:

```
${hash_3:monty}
${hash_5:monty}
${hash_4_62:monty python}
```

yield

```
jmg
monty
fbWx
```

respectively. The abbreviation **h** can be used instead of **hash**.

**${nhash_**<*n*>**:**<*string*>**}**

The string is processed by a hash function which returns a numeric value in the range 0–<*n*>-1.

**${nhash_**<*n*>**_**<*m*>**:**<*string*>**}**

The string is processed by a div/mod hash function which returns two numbers, separated by a slash, in the ranges 0–<*n*>-1 and 0–<*m*>-1, respectively. For example,

```
${nhash_8_64:supercalifragilisticexpialidocious}
```

returns the string ‘6/33’.

**${md5:**<*string*>**}**

The **md5** operator computes the MD5 hash value of the string, and returns it as a 32-digit hexadecimal number.

**${lc:**<*string*>**}**

This forces the letters in the string into lower-case, for example:

```
${lc:$local_part}
```

**${uc:**<*string*>**}**

This forces the letters in the string into upper-case.

**${length_**<*number*>**:**<*string*>**}**

The **length** operator can be used to extract the initial portion of a string. It is followed by an underscore and the number of characters required. For example

```
${length_50:$message_body}
```

The result of this operator is either the first <*number*> characters or the whole string, whichever is the shorter. The abbreviation **l** can be used instead of **length**.

**${local_part:**<*string*>**}**

The string is interpreted as an RFC 822 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, 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 fullstops (periods) to separate components instead of colons, because colon terminates a key string in lsearch files. So, for example,

```
${mask:5f03:1200:836f:0a00:000a:0800:200a:c031/99}
```

returns the string

```
5f03.1200.836f.0a00.000a.0800.2000.0000/99
```

Letters in IPv6 addresses are always output in lower case.

**${quote:**<*string*>**}**

The **quote** operator puts its argument into double quotes if it contains anything other than letters, digits, underscores, full stops (periods), and hyphens. Any occurrences of double quotes and backslashes are escaped with a backslash. 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_**<*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 6 of the main Exim specification. For example,

```
${quote_ldap:two + two}
```

returns ‘two%20%5C+%20two’. For single-key lookup types, no quoting is necessary and this operator yields an unchanged string.

**${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.

**${substr_**<*start*>**_**<*length*>**:**<*string*>**}**

The **substr** operator can be used to extract more general substrings than **length**. It is followed by an underscore and the starting offset, then a second underscore and 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 abbreviation **s** can be used instead of **substr**.

The **substr** expansion operator can take negative offset values to count from the righthand 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’.

If the second number is omitted from **substr**, the remainder of the string is taken if the offset was positive. If it was negative, all characters in the string preceding the offset point are taken. For example, an offset of -1 and no length yields all but the last character of the string.

## 32. 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.

**def:**<*variable name*>

The **def** condition must be followed by the name of one of the expansion variables defined in section 33. The condition is true if the named expansion 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 that no **$** appears before **header_** or **h_** in the condition, and that header names must be terminated by colons if white space does not follow.

**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.

**eq {**<*string1*>**}{**<*string2*>**}**

The two substrings are first expanded. The condition is true if the two resulting strings are identical, including the case of letters.

**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 with backslashes. Care must also be taken if the regular expression contains braces (curly brackets). A closing brace must be escaped so that it is not taken as a premature termination of <*string2*>. It does no harm to escape opening braces, but this is not strictly necessary. For example,

```
${if match {$local_part}{^\\d\{3\}} ...
```

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. 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.

**pam {**<*string1*>**:**<*string2*>**:...}**

*Pluggable Authentication Modules* (**http://www.kernel.org/pub/linux/libs/pam/**) are a facility which 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. 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 (i.e. <*string1*>). The remaining items in the data string are

passed over in response to requests from the authentication function. In the simple case there will only be one request, for a password, so the data will consist of 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{$1:${sg{$2}{:}{::}}}{yes}{no}}
```

**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.

**queue_running**

This condition, which has no data, is true during delivery attempts that are initiated by queue-runner processes, and false otherwise.

**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.

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.

### 33. Expansion variables

This list of expansion variable substitutions contains those that are likely to be of use in filter files. Others that are not relevant at filtering time, or are of interest only to the system administrator, are omitted.

**$0**, **$1**, etc: When a **matches** expansion condition succeeds, these variables contain the captured substrings identified by the regular expression during subsequent processing of the success string of the containing *if* expansion item. They may also be set externally by some other matching process 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.

**$domain**: When an address is being directed, routed, or delivered on its own, this variable contains the domain. In particular, it is set during user filtering, but not during system filtering, since a message may have many recipients and the system filter is called just once.

**$home**: This is set to the user's home directory when user filtering is configured in the normal way. When running a filter test via the **-bf** option, **$home** is set to the value of the environment variable HOME.

**$local_part**: When an address is being directed, routed, or delivered on its own, this variable contains the local part. If a local part prefix or suffix has been recognized, it is not included in the value.

**$local_part_prefix**: When an address is being directed or delivered locally, and a specific prefix for the local part was recognized, it is available in this variable. Otherwise it is empty.

**$local_part_suffix**: When an address is being directed or delivered locally, and a specific suffix for the local part was recognized, it is available in this variable. Otherwise it is empty.

**$message_body**: This variable contains the initial portion of a message's body while it is being delivered, and is intended mainly for use in filter files. The maximum number of characters of the body that are used is set by the **message_body_visible** configuration option; the default is 500. Newlines are converted into spaces to make it easier to search for phrases that might be split over a line break.

**$message_body_end**: This variable contains the final portion of a message's body while it is being delivered. The format and maximum size are as for **$message_body**.

**$message_body_size**: When a message is being received or delivered, this variable contains the size of the body in bytes. The count starts from the character after the blank line that separates the body from the header. Newlines are included in the count. See also **$message_size** and **$body_linecount**.

**$message_headers**: This variable contains a concatenation of all the header lines when a message is being processed. They are separated by newline characters.

**$message_id**: When a message is being received or delivered, this variable contains the unique message id which is used by Exim to identify the message.

**$message_precedence**: When a message is being delivered, the value of any **Precedence:** header is made available in this variable. If there is no such header, the value is the null string.

**$message_size**: When a message is being received or delivered, this variable contains its size in bytes. In most cases, the size includes those headers that were received with the message, but not those (such as **Envelope-to:**) that are added to individual deliveries as they are written. See also **$message_body_size** and **$body_linecount**.

**$n0** – **$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. This differs from **$parent_domain** when there is more than one level of aliasing or forwarding.

**$original_local_part**: When a top-level address is being processed for delivery, this contains the same value as **$local_part**. However, if a 'child' address (for example, generated by an alias, forward, or filter file) is being processed, this variable contains the local part of the original address. This differs from **$parent_local_part** when there is more than one level of aliasing or forwarding.

**$originator_gid**: The value of **$caller_gid** that was set when the message was received. For messages received via the command line, this is the gid of the sending user. For messages received by SMTP over TCP/IP, this is normally the gid of the Exim user.

**$originator_uid**: The value of **$caller_uid** that was set when the message was received. For messages received via the command line, this is the uid of the sending user. For messages received by SMTP over TCP/IP, this is normally the uid of the Exim user.

**$parent_domain**: This variable is empty, except when a 'child' address (generated by aliasing or forwarding, for example) is being processed, in which case it contains the domain of the immediately preceding parent address.

**$parent_local_part**: This variable is empty, except when a 'child' address (generated by aliasing or forwarding, for example) is being processed, in which case it contains the local part of the immediately preceding parent address.

**$primary_hostname**: The value set in the configuration file, or read by the *uname()* function.

**$qualify_domain**: The value set for this option in the configuration file.

**$qualify_recipient**: The value set for this option in the configuration file, or if not set, the value of **$qualify_domain**.

**$received_protocol**: When a message is being processed, this variable contains the name of the protocol by which it was received.

**$reply_address**: When a message is being processed, this variable contains the contents of the **Reply-To:** header line if one exists, or otherwise the contents of the **From:** header line. However, if the message contains a set of `Resent-` header lines, their contents are used in preference.

**$return_path**: When a message is being delivered, this variable contains the return path – the sender field that will be sent as part of the envelope. It is not enclosed in <> characters. In many cases, **$return_path** has the same value as **$sender_address**, but if, for example, an incoming message to a mailing list has been expanded by a director which specifies a specific address for delivery error messages, **$return_path** contains the new error address, while **$sender_address** contains the original sender address that was received with the message.

**$sender_address**: When a message is being processed, this variable contains the sender's address that was received in the message's envelope. For delivery failure reports, the value of this variable is the empty string.

**$sender_address_domain**: The domain portion of **$sender_address**.

**$sender_address_local_part**: The local part portion of **$sender_address**.

**$sender_fullhost**: When a message is received from a remote host, this variable contains the host name and IP address in a single string, which always ends with the IP address in square brackets. If **log_incoming_port** is set, the port number on the remote host is added to the IP address, separated by a full stop. 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. 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 first item in the argument of that command is placed in this variable. It is also set if HELO or EHLO is used when a message is received using SMTP locally via the **-bs** or **-bS** options.

**$sender_host_address**: When a message is received from a remote host, this variable contains that host's IP address.

**$sender_host_name**: When a message is received from a remote host, this variable contains the host's name as verified by looking up its IP address. If verification failed, or was not requested, this variable contains the empty string.

**$sender_host_port**: When a message is received from a remote host, this variable contains the port number that was used on the remote host.

**$sender_ident**: When a message 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.

**$sn0** – **$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.

**$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.

**$tod_bsdinbox**: The time of day and date, in the format required for BSD-style mailbox files, for example: Thu Oct 17 17:14:09 1995.

**$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 GMT.

**$tod_log**: The time and date in the format used for writing Exim's log files, for example: 1995-10-12 15:32:29.

**$value**: This variable contains the result of an expansion lookup or extraction operation, as described above. If **$value** is used in other circumstances, its contents are null.

**$version_number**: The version number of Exim.