

Section 11 - String expansions

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 be 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. 11.4 Forced expansion failure

A number of expansions that are described in the following section have alternative `&ldquo>true”` and `&ldquo>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 `&ldquo>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.

`$(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). `$(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
 abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

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>:

```

See `$rheader` below. `$bheader_<header name>:` or `$bh_<header name>:`

See `$rheader` below. `$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 warn 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 by warn statements 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. A newline character is inserted between each line. For the header expansion, for those headers that contain lists of addresses, a comma is also inserted at the junctions between lines. 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=coghheeLei2thahw`

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:gcoss} \`
`{ $value } fail }`
 `${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 keysting. 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 39.38. `#{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 39.38.

`#{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. ``${rheader_<header name>:}`` or ``${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.

If the command succeeds (gives a zero return code) `<string1>` is expanded and replaces the entire item; during this expansion, the standard output from the command is in the variable ``${value}``. If the command fails, `<string2>`, if present, is expanded and used. Once again, during the expansion, the standard output from the command is in the variable ``${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 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. `$(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 in expansion strings. The string (after expansion) must be a conventional arithmetic expression, but it is limited to five basic operators (plus, minus, times, divide, remainder) and parentheses. All operations are carried out using integer arithmetic. Plus and minus have a lower priority than times, divide, and remainder; operators with the same priority are evaluated from left to right.

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. 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
```

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. `$(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. `$(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. `$(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. `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 (also known as `bigcrypt()`), which was originally created to use up to 16 characters of the password. 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()`.

If you do not put any curly bracket encryption type in a `crypteq` comparison, the default is 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.

Note that if a password is no longer than 8 characters, the results of encrypting it with `crypt()` and `crypt16()` are identical. That means that `crypt16()` is backwards compatible, as long as nobody feeds it a password longer than 8 characters.

`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>}`

The two substrings are first expanded. The condition is true if the two resulting strings are identical, including the case of letters. `eqi {<string1>}{<string2>}`

The two substrings are first expanded. The condition is true if the two resulting strings are identical when compared in a case-independent way. `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. `ge {<string1>}{<string2>}`

See `gei`. `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>}`

See `gti`. `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>}`

See `isip6`. `isip4 {<string>}`

See `isip6`. `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 just for IPv4 or IPv6 addresses, respectively. For example, you could use `${if isip4{$sender_host_address}}...`

to test which version of IP 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>}`

See `lei`. `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>}`

See `lti`. `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.

-

Lookups are assumed to be `“net-”` style lookups, even if `net-` is not specified. Thus, the following are equivalent: `$(if match_ip{$sender_host_address}{|search;/some/file}...`
`$(if match_ip{$sender_host_address}{net-|search;/some/file}...`

You do need to specify the `net-` prefix if you want to specify a specific address mask, for example, by using `net24-`.

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{$1:$(sg{$2}{:}{:})}{yes}{no})`

For a PLAIN authenticator you could use: `server_condition = $(if pam{$2:$(sg{$3}{:}{:})}{yes}{no})`

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{$1:$2}{1}{0})`

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>}{yes}{no}}`
`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{{1}{2}}{1}{0}}`

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. 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. `$acl_c0` – `$acl_c19`

Values can be placed in these variables by the set modifier in an ACL. The values persist throughout the lifetime of an SMTP connection. They can be used to pass information between ACLs and 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. `$acl_m0` – `$acl_m19`

Values can be placed in these variables by the set modifier in an ACL. They retain their values while a message is being received, but are reset afterwards. They 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 `“/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 34–37). 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 `$authenticated_id` is the login name of the calling process. `$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`. `$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 45). `$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 45). `$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 40.6. `$demime_reason`

This variable is available when Exim is compiled with the content-scanning extension and the obsolete demime condition. For details, see section 40.6. `$dnslist_domain`

When a client host is found to be on a DNS (black) list, the list's domain name is put into this variable so that it can be included in the rejection message. `$dnslist_text`

When a client host is found to be on a DNS (black) list, the contents of any associated TXT record are placed in this variable. `$dnslist_value`

When a client host is found to be on a DNS (black) list, the IP address from the resource record is placed in this variable. If there are multiple records, all the addresses are included, comma-space separated. `$domain`

When an address is being routed, or delivered on its own, this variable contains the 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, `$domain` contains the complete argument of the ETRN command (see section 44.8). `$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 40.6. `$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`

When the smtp transport is expanding its options for encryption using TLS, `$host` contains the name of the host to which it is connected. Likewise, when used in the client part of an authenticator configuration (see chapter 33), `$host` contains the name of the server to which the client is connected.

When used in a transport filter (see chapter 24) `$host` refers to the host involved in the current connection. When a local transport is run as a result of a router that sets up a host list, `$host` contains the name of the first host. `$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`

As soon as a server starts processing a TCP/IP connection, this variable is set to the address of the local IP interface, and `$interface_port` is set to the port number. These values are therefore available for use in the "connect" ACL. See also the `-oMi` command line option. As well as being used in ACLs, these variable could be used, for example, to make the file name for a TLS certificate depend on which interface and/or port is being used. `$interface_port`

See `$interface_address`. `$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 and 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 41 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 40.1). `$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. Newlines are converted into spaces to make it easier to search for phrases that might be split over a line break. Binary zeros are also 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. `$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 `$mime` are available when Exim is compiled with the content-scanning extension. For details, see section 40.4. `$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 `$domain`. However, if a “child” address (for example, generated by an alias, forward, or filter file) is being processed, this variable contains the domain of the original address. This differs from `$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 39.38. `$prvscheck_keynum`

This variable is used in conjunction with the `prvscheck` expansion item, which is described in sections 11.5 and 39.38. `$prvscheck_result`

This variable is used in conjunction with the `prvscheck` expansion item, which is described in sections 11.5 and 39.38. `$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 (4xx) 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 (5xx) 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_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 `“esmt”`; (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 two cases:

-
- In a system filter file.
-

In the ACLs associated with the DATA command, that is, the ACLs defined by acl_smtp_predata and acl_smtp_data. \$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. \$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. 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 `$sender_host_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 10.15.

-

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 `$sender_rate_` are set as part of the ratelimit ACL condition. Details are given in section 39.30. `$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 `$recipient_verify_failure`. `$smtp_active_hostname`

During an 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 `$smtp_command`, this variable is somewhat redundant, but is retained for backwards compatibility. `$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 `$spam` are available when Exim is compiled with the content-scanning extension. For details, see section 40.2. `$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. See chapter 38 for details of TLS support. `$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 `$tls_peerdn` during subsequent processing. `$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 `yyymmdd`. 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. `$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 45.2. `$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 45.2.