

Section 20 - The manualroute router

20. The manualroute router

The manualroute router is so-called because it provides a way of manually routing an address according to its domain. It is mainly used when you want to route addresses to remote hosts according to your own rules, bypassing the normal DNS routing that looks up MX records. However, manualroute can also route to local transports, a facility that may be useful if you want to save messages for dial-in hosts in local files.

The manualroute router compares a list of domain patterns with the domain it is trying to route. If there is no match, the router declines. Each pattern has associated with it a list of hosts and some other optional data, which may include a transport. The combination of a pattern and its data is called a "routing rule". For patterns that do not have an associated transport, the generic transport option must specify a transport, unless the router is being used purely for verification (see `verify_only`).

In the case of verification, matching the domain pattern is sufficient for the router to accept the address. When actually routing an address for delivery, an address that matches a domain pattern is queued for the associated transport. If the transport is not a local one, a host list must be associated with the pattern; IP addresses are looked up for the hosts, and these are passed to the transport along with the mail address. For local transports, a host list is optional. If it is present, it is passed in `$host` as a single text string.

The list of routing rules can be provided as an inline string in `route_list`, or the data can be obtained by looking up the domain in a file or database by setting `route_data`. Only one of these settings may appear in any one instance of manualroute. The format of routing rules is described below, following the list of private options. 20.1 Private options for manualroute

The private options for the manualroute router are as follows:

`host_find_failedUse: manualrouteType: stringDefault: freeze`

This option controls what happens when manualroute tries to find an IP address for a host, and the host does not exist. The option can be set to one of `decline`
`defer`
`fail`
`freeze`
`pass`

The default assumes that this state is a serious configuration error. The difference between "pass" and "decline" is that the former forces the address to be passed to the next router (or the router defined by `pass_router`), overriding `no_more`, whereas the latter passes the address to the next router only if more is true.

This option applies only to a definite "does not exist" state; if a host lookup gets a temporary error, delivery is deferred unless the generic `pass_on_timeout` option is set.

`hosts_randomizeUse: manualrouteType: booleanDefault: false`

If this option is set, the order of the items in a host list in a routing rule is randomized each time the list is used, unless an option in the routing rule overrides (see below). Randomizing the order of a host list can be used to do crude load sharing. However, if more than one mail address is routed by the same router to the same host list, the host lists are considered to be the same (even though they may be randomized into different orders) for the purpose of deciding whether to batch the deliveries into a single SMTP transaction.

When `hosts_randomize` is true, a host list may be split into groups whose order is separately randomized. This makes it possible to set up MX-like behaviour. The boundaries between groups are indicated by an item that is just `+` in the host list. For example: `route_list = * host1:host2:host3+:host4:host5`

The order of the first three hosts and the order of the last two hosts is randomized for each use, but the first three always end up before the last two. If `hosts_randomize` is not set, a `+` item in the list is ignored. If a randomized host list is passed to an `smtp` transport that also has `hosts_randomize` set, the list is not re-randomized.

`route_dataUse: manualrouteType: string†Default: unset`

If this option is set, it must expand to yield the data part of a routing rule. Typically, the expansion string includes a lookup based on the domain. For example: `route_data = ${lookup{$domain}dbm{/etc/routes}}`

If the expansion is forced to fail, or the result is an empty string, the router declines. Other kinds of expansion failure cause delivery to be deferred.

`route_listUse: manualrouteType: Default: string`

This string is a list of routing rules, in the form defined below. Note that, unlike most string lists, the items are separated by semicolons. This is so that they may contain colon-separated host lists.

`same_domain_copy_routingUse: manualrouteType: booleanDefault: false`

Addresses with the same domain are normally routed by the manualroute router to the same list of hosts. However, this cannot be presumed, because the router options and preconditions may refer to the local part of the address. By default, therefore, Exim routes each address in a message independently. DNS servers run caches, so repeated DNS lookups are not normally expensive, and in any case, personal messages rarely have more than a few recipients.

If you are running mailing lists with large numbers of subscribers at the same domain, and you are using a manualroute router which is independent of the local part, you can set `same_domain_copy_routing` to bypass repeated DNS lookups for identical domains in one message. In this case, when manualroute routes an address to a remote transport, any other unrouted addresses in the message that have the same domain are automatically given the same routing without processing them independently. However, this is only done if `headers_add` and `headers_remove` are unset.

20.2 Routing rules in route_list

The value of `route_list` is a string consisting of a sequence of routing rules, separated by semicolons. If a semicolon is needed in a rule, it can be entered as two semicolons. Alternatively, the list separator can be changed as described (for colon-separated lists) in section 6.19. Empty rules are ignored. The format of each rule is `<domain pattern> <list of hosts> <options>`

The following example contains two rules, each with a simple domain pattern and no options: `route_list = \dict.ref.example mail-1.ref.example:mail-2.ref.example ; \thes.ref.example mail-3.ref.example:mail-4.ref.example`

The three parts of a rule are separated by white space. The pattern and the list of hosts can be enclosed in quotes if necessary, and if they are, the usual quoting rules apply. Each rule in a `route_list` must start with a single domain pattern, which is the only mandatory item in the rule. The pattern is in the same format as one item in a domain list (see section 10.8), except that it may not be the name of an interpolated file. That is, it may be wildcarded, or a regular expression, or a file or database lookup (with semicolons doubled, because of the use of semicolon as a separator in a `route_list`).

The rules in `route_list` are searched in order until one of the patterns matches the domain that is being routed. The list of hosts and then options are then used as described below. If there is no match, the router declines. When `route_list` is set, `route_data` must not be set.

20.3 Routing rules in route_data

The use of `route_list` is convenient when there are only a small number of routing rules. For larger numbers, it is easier to use a file or database to hold the routing information, and use the `route_data` option instead. The value of `route_data` is a list of hosts, followed by (optional) options. Most commonly, `route_data` is set as a string that contains an expansion lookup. For example, suppose we place two routing rules in a file like this: `dict.ref.example: mail-1.ref.example:mail-2.ref.example`
`thes.ref.example: mail-3.ref.example:mail-4.ref.example`

This data can be accessed by setting `route_data = ${lookup{$domain}search{/the/file/name}}`

Failure of the lookup results in an empty string, causing the router to decline. However, you do not have to use a lookup in `route_data`. The only requirement is that the result of expanding the string is a list of hosts, possibly followed by options, separated by white space. The list of hosts must be enclosed in quotes if it contains white space.

20.4 Format of the list of hosts

A list of hosts, whether obtained via `route_data` or `route_list`, is always separately expanded before use. If the expansion fails, the router declines. The result of the expansion must be a colon-separated list of names and/or IP addresses,

optionally also including ports. The format of each item in the list is described in the next section. The list separator can be changed as described in section 6.19.

If the list of hosts was obtained from a `route_list` item, the following variables are set during its expansion:

-

If the domain was matched against a regular expression, the numeric variables `$1`, `$2`, etc. may be set. For example: `route_list = ^domain(\d+) host-$1.text.example`

-

`$0` is always set to the entire domain.

-

`$1` is also set when partial matching is done in a file lookup.

-

If the pattern that matched the domain was a lookup item, the data that was looked up is available in the expansion variable `$value`. For example: `route_list = lsearch;/some/file.routes $value`

Note the doubling of the semicolon in the pattern that is necessary because semicolon is the default route list separator.

20.5 Format of one host item

Each item in the list of hosts is either a host name or an IP address, optionally with an attached port number. When no port is given, an IP address is not enclosed in brackets. When a port is specified, it overrides the port specification on the transport. The port is separated from the name or address by a colon. This leads to some complications:

-

Because colon is the default separator for the list of hosts, either the colon that specifies a port must be doubled, or the list separator must be changed. The following two examples have the same effect: `route_list = * "host1.tld::1225 : host2.tld::1226"`

`route_list = * "<+ host1.tld:1225 + host2.tld:1226"`

-

When IPv6 addresses are involved, it gets worse, because they contain colons of their own. To make this case easier, it is permitted to enclose an IP address (either v4 or v6) in square brackets if a port number follows. For example:

`route_list = * "</ [10.1.1.1]:1225 / [::1]:1226"`

20.6 How the list of hosts is used

When an address is routed to an smtp transport by `manualroute`, each of the hosts is tried, in the order specified, when carrying out the SMTP delivery. However, the order can be changed by setting the `hosts_randomize` option, either on the router (see section 20.1 above), or on the transport.

Hosts may be listed by name or by IP address. An unadorned name in the list of hosts is interpreted as a host name. A name that is followed by `/MX` is interpreted as an indirection to a sublist of hosts obtained by looking up MX records in the DNS. For example: `route_list = * x.y.z.p.q.r/MX:e.f.g`

If this feature is used with a port specifier, the port must come last. For example: `route_list = * dom1.tld/mx::1225`

If the `hosts_randomize` option is set, the order of the items in the list is randomized before any lookups are done. Exim then scans the list; for any name that is not followed by `/MX` it looks up an IP address. If this turns out to be an interface on the local host and the item is not the first in the list, Exim discards it and any subsequent items. If it is the first item, what happens is controlled by the `self` option of the router.

A name on the list that is followed by `/MX` is replaced with the list of hosts obtained by looking up MX records for the name. This is always a DNS lookup; the `bydns` and `byname` options (see section 20.7 below) are not relevant here. The order of these hosts is determined by the preference values in the MX records, according to the usual rules. Because randomizing happens before the MX lookup, it does not affect the order that is defined by MX preferences.

If the local host is present in the sublist obtained from MX records, but is not the most preferred host in that list, it and any equally or less preferred hosts are removed before the sublist is inserted into the main list.

If the local host is the most preferred host in the MX list, what happens depends on where in the original list of hosts the /MX item appears. If it is not the first item (that is, there are previous hosts in the main list), Exim discards this name and any subsequent items in the main list.

If the MX item is first in the list of hosts, and the local host is the most preferred host, what happens is controlled by the self option of the router.

DNS failures when lookup up the MX records are treated in the same way as DNS failures when looking up IP addresses: `pass_on_timeout` and `host_find_failed` are used when relevant.

The generic `ignore_target_hosts` option applies to all hosts in the list, whether obtained from an MX lookup or not.

20.7 How the options are used

The options are a sequence of words; in practice no more than three are ever present. One of the words can be the name of a transport; this overrides the transport option on the router for this particular routing rule only. The other words (if present) control randomization of the list of hosts on a per-rule basis, and how the IP addresses of the hosts are to be found when routing to a remote transport. These options are as follows:

-
- `randomize`: randomize the order of the hosts in this list, overriding the setting of `hosts_randomize` for this routing rule only.

-
- `no_randomize`: do not randomize the order of the hosts in this list, overriding the setting of `hosts_randomize` for this routing rule only.

-
- `byname`: use `getipnodebyname()` (`gethostbyname()` on older systems) to find IP addresses. This function may ultimately cause a DNS lookup, but it may also look in `/etc/hosts` or other sources of information.

-
- `bydns`: look up address records for the hosts directly in the DNS; fail if no address records are found. If there is a temporary DNS error (such as a timeout), delivery is deferred.

For example: `route_list = domain1 host1:host2:host3 randomize bydns;\`
`domain2 host4:host5`

If neither `byname` nor `bydns` is given, Exim behaves as follows: First, a DNS lookup is done. If this yields anything other than `HOST_NOT_FOUND`, that result is used. Otherwise, Exim goes on to try a call to `getipnodebyname()` or `gethostbyname()`, and the result of the lookup is the result of that call.

Warning: It has been discovered that on some systems, if a DNS lookup called via `getipnodebyname()` times out, `HOST_NOT_FOUND` is returned instead of `TRY_AGAIN`. That is why the default action is to try a DNS lookup first. Only if that gives a definite "no such host" is the local function called.

If no IP address for a host can be found, what happens is controlled by the `host_find_failed` option.

When an address is routed to a local transport, IP addresses are not looked up. The host list is passed to the transport in the `$host` variable.

20.8 Manualroute examples

In some of the examples that follow, the presence of the `remote_smtp` transport, as defined in the default configuration file, is assumed:

-
- The `manualroute` router can be used to forward all external mail to a smart host. If you have set up, in the main part of the configuration, a named domain list that contains your local domains, for example: `domainlist local_domains = my.domain.example`

You can arrange for all other domains to be routed to a smart host by making your first router something like this:

```
smart_route:
  driver = manualroute
  domains = !+local_domains
```

```
transport = remote_smtp
route_list = * smarthost.ref.example
```

This causes all non-local addresses to be sent to the single host `smarthost.ref.example`. If a colon-separated list of smart hosts is given, they are tried in order (but you can use `hosts_randomize` to vary the order each time). Another way of configuring the same thing is this: `smart_route`:

```
driver = manualroute
transport = remote_smtp
route_list = !+local_domains smarthost.ref.example
```

There is no difference in behaviour between these two routers as they stand. However, they behave differently if `no_more` is added to them. In the first example, the router is skipped if the domain does not match the domains precondition; the following router is always tried. If the router runs, it always matches the domain and so can never decline. Therefore, `no_more` would have no effect. In the second case, the router is never skipped; it always runs. However, if it doesn't match the domain, it declines. In this case `no_more` would prevent subsequent routers from running.

A mail hub is a host which receives mail for a number of domains via MX records in the DNS and delivers it via its own private routing mechanism. Often the final destinations are behind a firewall, with the mail hub being the one machine that can connect to machines both inside and outside the firewall. The `manualroute` router is usually used on a mail hub to route incoming messages to the correct hosts. For a small number of domains, the routing can be inline, using the `route_list` option, but for a larger number a file or database lookup is easier to manage.

If the domain names are in fact the names of the machines to which the mail is to be sent by the mail hub, the configuration can be quite simple. For example: `hub_route`:

```
driver = manualroute
transport = remote_smtp
route_list = *.rhodes.tvs.example $domain
```

This configuration routes domains that match `*.rhodes.tvs.example` to hosts whose names are the same as the mail domains. A similar approach can be taken if the host name can be obtained from the domain name by a string manipulation that the expansion facilities can handle. Otherwise, a lookup based on the domain can be used to find the host: `through_firewall`:

```
driver = manualroute
transport = remote_smtp
route_data = ${lookup {$domain} cdb {/internal/host/routes}}
```

The result of the lookup must be the name or IP address of the host (or hosts) to which the address is to be routed. If the lookup fails, the route data is empty, causing the router to decline. The address then passes to the next router.

You can use `manualroute` to deliver messages to pipes or files in batched SMTP format for onward transportation by some other means. This is one way of storing mail for a dial-up host when it is not connected. The route list entry can be as simple as a single domain name in a configuration like this: `save_in_file`:

```
driver = manualroute
transport = batchsmtp_appendfile
route_list = saved.domain.example
```

though often a pattern is used to pick up more than one domain. If there are several domains or groups of domains with different transport requirements, different transports can be listed in the routing information: `save_in_file`:

```
driver = manualroute
route_list = \
*.saved.domain1.example $domain batch_appendfile; \
*.saved.domain2.example \
${lookup{$domain}dbm{/domain2/hosts}{$value}fail} \
batch_pipe
```

The first of these just passes the domain in the `$host` variable, which doesn't achieve much (since it is also in

\$domain), but the second does a file lookup to find a value to pass, causing the router to decline to handle the address if the lookup fails.

-

Routing mail directly to UUCP software is a specific case of the use of manualroute in a gateway to another mail environment. This is an example of one way it can be done: # Transport

```
uucp:
driver = pipe
user = nobody
command = /usr/local/bin/uux -r - \
  ${substr_-5:$host}!rmail ${local_part}
return_fail_output = true
```

```
# Router
uucphost:
transport = uucp
driver = manualroute
route_data = \
  ${lookup{$domain}!search{/usr/local/exim/uucphosts}}
```

The file /usr/local/exim/uucphosts contains entries like darksite.ethereal.example: darksite.UUCP

It can be set up more simply without adding and removing "UUCP"; but this way makes clear the distinction between the domain name darksite.ethereal.example and the UUCP host name darksite.