

# BIRD User's Guide

---

Ondrej Filip <[feela@network.cz](mailto:feela@network.cz)>, Pavel Machek <[pavel@ucw.cz](mailto:pavel@ucw.cz)>, Martin Mares <[mj@ucw.cz](mailto:mj@ucw.cz)>, Ondrej Zajicek <[santiago@crfreenet.org](mailto:santiago@crfreenet.org)>

This document contains user documentation for the BIRD Internet Routing Daemon project.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	What is BIRD	3
1.2	Installing BIRD	4
1.3	Running BIRD	4
1.4	Privileges	5
<b>2</b>	<b>About routing tables</b>	<b>6</b>
2.1	Graceful restart	6
<b>3</b>	<b>Configuration</b>	<b>7</b>
3.1	Introduction	7
3.2	Global options	7
3.3	Protocol options	9
<b>4</b>	<b>Remote control</b>	<b>13</b>
<b>5</b>	<b>Filters</b>	<b>16</b>
5.1	Introduction	16
5.2	Data types	17
5.3	Operators	20
5.4	Control structures	20
5.5	Route attributes	20
5.6	Other statements	22
<b>6</b>	<b>Protocols</b>	<b>23</b>
6.1	Babel	23
6.2	BFDD	24
6.3	BGP	27
6.4	Device	35
6.5	Direct	36
6.6	Kernel	36
6.7	MRT	38
6.8	OSPF	39
6.9	Pipe	47
6.10	RAdv	49
6.11	RIP	54
6.12	Static	57
<b>7</b>	<b>Conclusions</b>	<b>60</b>
7.1	Future work	60
7.2	Getting more help	60

# Chapter 1: Introduction

## 1.1 What is BIRD

The name ‘BIRD’ is actually an acronym standing for ‘BIRD Internet Routing Daemon’. Let’s take a closer look at the meaning of the name:

*BIRD*: Well, we think we have already explained that. It’s an acronym standing for ‘BIRD Internet Routing Daemon’, you remember, don’t you? :-)

*Internet Routing*: It’s a program (well, a daemon, as you are going to discover in a moment) which works as a dynamic router in an Internet type network (that is, in a network running either the IPv4 or the IPv6 protocol). Routers are devices which forward packets between interconnected networks in order to allow hosts not connected directly to the same local area network to communicate with each other. They also communicate with the other routers in the Internet to discover the topology of the network which allows them to find optimal (in terms of some metric) rules for forwarding of packets (which are called routing tables) and to adapt themselves to the changing conditions such as outages of network links, building of new connections and so on. Most of these routers are costly dedicated devices running obscure firmware which is hard to configure and not open to any changes (on the other hand, their special hardware design allows them to keep up with lots of high-speed network interfaces, better than general-purpose computer does). Fortunately, most operating systems of the UNIX family allow an ordinary computer to act as a router and forward packets belonging to the other hosts, but only according to a statically configured table.

A *Routing Daemon* is in UNIX terminology a non-interactive program running on background which does the dynamic part of Internet routing, that is it communicates with the other routers, calculates routing tables and sends them to the OS kernel which does the actual packet forwarding. There already exist other such routing daemons: routed (RIP only), GateD (non-free), [Zebra](#) and [MRTD](#), but their capabilities are limited and they are relatively hard to configure and maintain.

BIRD is an Internet Routing Daemon designed to avoid all of these shortcomings, to support all the routing technology used in the today’s Internet or planned to be used in near future and to have a clean extensible architecture allowing new routing protocols to be incorporated easily. Among other features, BIRD supports:

- both IPv4 and IPv6 protocols
- multiple routing tables
- the Border Gateway Protocol (BGPv4)
- the Routing Information Protocol (RIPv2)
- the Open Shortest Path First protocol (OSPFv2, OSPFv3)
- the Router Advertisements for IPv6 hosts
- a virtual protocol for exchange of routes between different routing tables on a single host
- a command-line interface allowing on-line control and inspection of status of the daemon
- soft reconfiguration (no need to use complex online commands to change the configuration, just edit the configuration file and notify BIRD to re-read it and it will smoothly switch itself to the new configuration, not disturbing routing protocols unless they are affected by the configuration changes)
- a powerful language for route filtering

BIRD has been developed at the Faculty of Math and Physics, Charles University, Prague, Czech Republic as a student project. It can be freely distributed under the terms of the GNU General Public License.

BIRD has been designed to work on all UNIX-like systems. It has been developed and tested under Linux 2.0 to 2.6, and then ported to FreeBSD, NetBSD and OpenBSD, porting to other systems (even non-UNIX ones) should be relatively easy due to its highly modular architecture.

BIRD supports either IPv4 or IPv6 protocol, but have to be compiled separately for each one. Therefore, a dualstack router would run two instances of BIRD (one for IPv4 and one for IPv6), with completely separate setups (configuration files, tools ...).

## 1.2 Installing BIRD

On a recent UNIX system with GNU development tools (GCC, binutils, m4, make) and Perl, installing BIRD should be as easy as:

```
./configure
make
make install
vi /usr/local/etc/bird.conf
bird
```

You can use `./configure --help` to get a list of configure options. The most important ones are: `--enable-ipv6` which enables building of an IPv6 version of BIRD, `--with-protocols=` to produce a slightly smaller BIRD executable by configuring out routing protocols you don't use, and `--prefix=` to install BIRD to a place different from `/usr/local`.

## 1.3 Running BIRD

You can pass several command-line options to `bird`:

- `-c config name`  
use given configuration file instead of `prefix/etc/bird.conf`.
- `-d`  
enable debug messages and run `bird` in foreground.
- `-D filename of debug log`  
log debugging information to given file instead of `stderr`.
- `-f`  
run `bird` in foreground.
- `-g group`  
use that group ID, see the next section for details.
- `-h, --help`  
display command-line options to `bird`.
- `-l`  
look for a configuration file and a communication socket in the current working directory instead of in default system locations. However, paths specified by options `-c`, `-s` have higher priority.
- `-p`  
just parse the config file and exit. Return value is zero if the config file is valid, nonzero if there are some errors.
- `-P name of PID file`  
create a PID file with given filename.
- `-R`  
apply graceful restart recovery after start.
- `-s name of communication socket`  
use given filename for a socket for communications with the client, default is `prefix/var/run/birdctl`.
- `-u user`  
drop privileges and use that user ID, see the next section for details.
- `--version`  
display `bird` version.

BIRD writes messages about its work to log files or syslog (according to config).

## 1.4 Privileges

BIRD, as a routing daemon, uses several privileged operations (like setting routing table and using raw sockets). Traditionally, BIRD is executed and runs with root privileges, which may be prone to security problems. The recommended way is to use a privilege restriction (options `-u`, `-g`). In that case BIRD is executed with root privileges, but it changes its user and group ID to an unprivileged ones, while using Linux capabilities to retain just required privileges (capabilities `CAP_NET_*`). Note that the control socket is created before the privileges are dropped, but the config file is read after that. The privilege restriction is not implemented in BSD port of BIRD.

An unprivileged user (as an argument to `-u` options) may be the user `nobody`, but it is suggested to use a new dedicated user account (like `bird`). The similar considerations apply for the group option, but there is one more condition – the users in the same group can use `birdc` to control BIRD.

Finally, there is a possibility to use external tools to run BIRD in an environment with restricted privileges. This may need some configuration, but it is generally easy – BIRD needs just the standard library, privileges to read the config file and create the control socket and the `CAP_NET_*` capabilities.

# Chapter 2: About routing tables

BIRD has one or more routing tables which may or may not be synchronized with OS kernel and which may or may not be synchronized with each other (see the Pipe protocol). Each routing table contains a list of known routes. Each route consists of:

- network prefix this route is for (network address and prefix length – the number of bits forming the network part of the address; also known as a netmask)
- preference of this route
- IP address of router which told us about this route
- IP address of router we should forward the packets to using this route
- other attributes common to all routes
- dynamic attributes defined by protocols which may or may not be present (typically protocol metrics)

Routing table maintains multiple entries for a network, but at most one entry for one network and one protocol. The entry with the highest preference is used for routing (we will call such an entry the *selected route*). If there are more entries with the same preference and they are from the same protocol, the protocol decides (typically according to metrics). If they aren't, an internal ordering is used to break the tie. You can get the list of route attributes in the Route attributes section.

Each protocol is connected to a routing table through two filters which can accept, reject and modify the routes. An *export* filter checks routes passed from the routing table to the protocol, an *import* filter checks routes in the opposite direction. When the routing table gets a route from a protocol, it recalculates the selected route and broadcasts it to all protocols connected to the table. The protocols typically send the update to other routers in the network. Note that although most protocols are interested in receiving just selected routes, some protocols (e.g. the Pipe protocol) receive and process all entries in routing tables (accepted by filters).

Usually, a routing table just chooses a selected route from a list of entries for one network. But if the `sorted` option is activated, these lists of entries are kept completely sorted (according to preference or some protocol-dependent metric). This is needed for some features of some protocols (e.g. `secondary` option of BGP protocol, which allows to accept not just a selected route, but the first route (in the sorted list) that is accepted by filters), but it is incompatible with some other features (e.g. `deterministic med` option of BGP protocol, which activates a way of choosing selected route that cannot be described using comparison and ordering). Minor advantage is that routes are shown sorted in `show route`, minor disadvantage is that it is slightly more computationally expensive.

## 2.1 Graceful restart

When BIRD is started after restart or crash, it repopulates routing tables in an uncoordinated manner, like after clean start. This may be impractical in some cases, because if the forwarding plane (i.e. kernel routing tables) remains intact, then its synchronization with BIRD would temporarily disrupt packet forwarding until protocols converge. Graceful restart is a mechanism that could help with this issue. Generally, it works by starting protocols and letting them repopulate routing tables while deferring route propagation until protocols acknowledge their convergence. Note that graceful restart behavior have to be configured for all relevant protocols and requires protocol-specific support (currently implemented for Kernel and BGP protocols), it is activated for particular boot by option `-R`.

# Chapter 3: Configuration

## 3.1 Introduction

BIRD is configured using a text configuration file. Upon startup, BIRD reads *prefix/etc/bird.conf* (unless the `-c` command line option is given). Configuration may be changed at user's request: if you modify the config file and then signal BIRD with `SIGHUP`, it will adjust to the new config. Then there's the client which allows you to talk with BIRD in an extensive way.

In the config, everything on a line after `#` or inside `/* */` is a comment, whitespace characters are treated as a single space. If there's a variable number of options, they are grouped using the `{ }` brackets. Each option is terminated by a `;`. Configuration is case sensitive. There are two ways how to name symbols (like protocol names, filter names, constants etc.). You can either use a simple string starting with a letter followed by any combination of letters and numbers (e.g. "R123", "myfilter", "bgp5") or you can enclose the name into apostrophes (`'`) and then you can use any combination of numbers, letters, hyphens, dots and colons (e.g. "'1:strange-name'", "'-NAME-'", "'cool:name'").

Here is an example of a simple config file. It enables synchronization of routing tables with OS kernel, scans for new network interfaces every 10 seconds and runs RIP on all network interfaces found.

```
protocol kernel {
    persist;                # Don't remove routes on BIRD shutdown
    scan time 20;           # Scan kernel routing table every 20 seconds
    export all;             # Default is export none
}

protocol device {
    scan time 10;          # Scan interfaces every 10 seconds
}

protocol rip {
    export all;
    import all;
    interface "*";
}
```

## 3.2 Global options

`include "filename"`

This statement causes inclusion of a new file. *Filename* could also be a wildcard, in that case matching files are included in alphabetic order. The maximal depth is 8. Note that this statement could be used anywhere in the config file, not just as a top-level option.

`log "filename"|syslog [name name]|stderr all|{ list of classes }`

Set logging of messages having the given class (either `all` or `{ error|trace [, ...] }` etc.) into selected destination (a file specified as a filename string, `syslog` with optional name argument, or the `stderr` output). Classes are: `info`, `warning`, `error` and `fatal` for messages about local problems, `debug` for debugging messages, `trace` when you want to know what happens in the network, `remote` for messages about misbehavior of remote machines, `auth` about authentication failures, `bug` for internal BIRD bugs. You may specify more than one `log` line to establish logging to multiple destinations. Default: log everything to the system log.

`debug protocols all|off|{ states|routes|filters|interfaces|events|packets [, ...] }`

Set global defaults of protocol debugging options. See `debug` in the following section. Default: `off`.

`debug commands number`

Control logging of client connections (0 for no logging, 1 for logging of connects and disconnects, 2 and higher for logging of all client commands). Default: 0.

**debug latency** *switch*

Activate tracking of elapsed time for internal events. Recent events could be examined using **dump events** command. Default: off.

**debug latency limit** *time*

If **debug latency** is enabled, this option allows to specify a limit for elapsed time. Events exceeding the limit are logged. Default: 1 s.

**watchdog warning** *time*

Set time limit for I/O loop cycle. If one iteration took more time to complete, a warning is logged. Default: 5 s.

**watchdog timeout** *time*

Set time limit for I/O loop cycle. If the limit is breached, BIRD is killed by abort signal. The timeout has effective granularity of seconds, zero means disabled. Default: disabled (0).

**mrtdump** "*filename*"

Set MRTdump file name. This option must be specified to allow MRTdump feature. Default: no dump file.

**mrtdump protocols** *all|off*{ *states|messages* [, ...] }

Set global defaults of MRTdump options. See **mrtdump** in the following section. Default: off.

**filter** *name local variables*{ *commands* }

Define a filter. You can learn more about filters in the following chapter.

**function** *name (parameters) local variables* { *commands* }

Define a function. You can learn more about functions in the following chapter.

**protocol** *rip|ospf|bgp|...* [*name* [*from name2*]] { *protocol options* }

Define a protocol instance called *name* (or with a name like "rip5" generated automatically if you don't specify any *name*). You can learn more about configuring protocols in their own chapters. When *from name2* expression is used, initial protocol options are taken from protocol or template *name2*. You can run more than one instance of most protocols (like RIP or BGP). By default, no instances are configured.

**template** *rip|bgp|...* [*name* [*from name2*]] { *protocol options* }

Define a protocol template instance called *name* (or with a name like "bgp1" generated automatically if you don't specify any *name*). Protocol templates can be used to group common options when many similarly configured protocol instances are to be defined. Protocol instances (and other templates) can use templates by using *from* expression and the name of the template. At the moment templates (and *from* expression) are not implemented for OSPF protocol.

**define** *constant = expression*

Define a constant. You can use it later in every place you could use a value of the same type. Besides, there are some predefined numeric constants based on */etc/iproute2/route.\** files. A list of defined constants can be seen (together with other symbols) using 'show symbols' command.

**router id** *IPv4 address*

Set BIRD's router ID. It's a world-wide unique identification of your router, usually one of router's IPv4 addresses. Default: in IPv4 version, the lowest IP address of a non-loopback interface. In IPv6 version, this option is mandatory.

**router id from** [-] [ "*mask*" ] [ *prefix* ] [, ...]

Set BIRD's router ID based on an IP address of an interface specified by an interface pattern. The option is applicable for IPv4 version only. See [interface](#) (p.11) section for detailed description of interface patterns with extended clauses.

**listen** *bgp* [*address address*] [*port port*] [*dual*]

This option allows to specify address and port where BGP protocol should listen. It is global option as listening socket is common to all BGP instances. Default is to listen on all addresses (0.0.0.0) and port 179. In IPv6 mode, option *dual* can be used to specify that BGP socket should accept both IPv4

and IPv6 connections (but even in that case, BIRD would accept IPv6 routes only). Such behavior was default in older versions of BIRD.

**graceful restart wait** *number*

During graceful restart recovery, BIRD waits for convergence of routing protocols. This option allows to specify a timeout for the recovery to prevent waiting indefinitely if some protocols cannot converge. Default: 240 seconds.

**timeformat** *route|protocol|base|log "format1" [limit "format2"]*

This option allows to specify a format of date/time used by BIRD. The first argument specifies for which purpose such format is used. **route** is a format used in 'show route' command output, **protocol** is used in 'show protocols' command output, **base** is used for other commands and **log** is used in a log file.

"*format1*" is a format string using *strftime(3)* notation (see *man strftime* for details). *limit*> and "*format2*" allow to specify the second format string for times in past deeper than *limit* seconds. There are few shorthands: **iso long** is a ISO 8601 datetime format (YYYY-MM-DD hh:mm:ss) that can be also specified using "%F %T". **iso short** is a variant of ISO 8601 that uses just the time format (hh:mm:ss) for near times (up to 20 hours in the past) and the date format (YYYY-MM-DD) for far times. This is a shorthand for "%T" 72000 "%F".

By default, BIRD uses the **iso short** format for **route** and **protocol** times, and the **iso long** format for **base** and **log** times.

In pre-1.4.0 versions, BIRD used an short, ad-hoc format for **route** and **protocol** times, and a **iso long** similar format (DD-MM-YYYY hh:mm:ss) for **base** and **log**. These timeformats could be set by **old short** and **old long** compatibility shorthands.

**table** *name* [sorted]

Create a new routing table. The default routing table is created implicitly, other routing tables have to be added by this command. Option **sorted** can be used to enable sorting of routes, see [sorted table](#) (p.6) description for details.

**roa table** *name* [ { *roa table options ...* } ]

Create a new ROA (Route Origin Authorization) table. ROA tables can be used to validate route origination of BGP routes. A ROA table contains ROA entries, each consist of a network prefix, a max prefix length and an AS number. A ROA entry specifies prefixes which could be originated by that AS number. ROA tables could be filled with data from RPKI ([RFC 6480](#)) or from public databases like Whois. ROA tables are examined by `roa_check()` operator in filters.

Currently, there is just one option, `roa prefix max num as num`, which can be used to populate the ROA table with static ROA entries. The option may be used multiple times. Other entries can be added dynamically by `add roa` command.

**eval** *expr*

Evaluates given filter expression. It is used by us for testing of filters.

### 3.3 Protocol options

For each protocol instance, you can configure a bunch of options. Some of them (those described in this section) are generic, some are specific to the protocol (see sections talking about the protocols).

Several options use a *switch* argument. It can be either **on**, **yes** or a numeric expression with a non-zero value for the option to be enabled or **off**, **no** or a numeric expression evaluating to zero to disable it. An empty *switch* is equivalent to **on** ("silence means agreement").

**preference** *expr*

Sets the preference of routes generated by this protocol. Default: protocol dependent.

**disabled** *switch*

Disables the protocol. You can change the disable/enable status from the command line interface without needing to touch the configuration. Disabled protocols are not activated. Default: protocol is enabled.

`debug all|off|{ states|routes|filters|interfaces|events|packets [ , ...] }`

Set protocol debugging options. If asked, each protocol is capable of writing trace messages about its work to the log (with category `trace`). You can either request printing of `all` trace messages or only of the types selected: `states` for protocol state changes (protocol going up, down, starting, stopping etc.), `routes` for routes exchanged with the routing table, `filters` for details on route filtering, `interfaces` for interface change events sent to the protocol, `events` for events internal to the protocol and `packets` for packets sent and received by the protocol. Default: `off`.

`mrtdump all|off|{ states|messages [ , ...] }`

Set protocol MRTdump flags. MRTdump is a standard binary format for logging information from routing protocols and daemons. These flags control what kind of information is logged from the protocol to the MRTdump file (which must be specified by global `mrtdump` option, see the previous section). Although these flags are similar to flags of `debug` option, their meaning is different and protocol-specific. For BGP protocol, `states` logs BGP state changes and `messages` logs received BGP messages. Other protocols does not support MRTdump yet.

`router id IPv4 address`

This option can be used to override global router id for a given protocol. Default: uses global router id.

`import all | none | filter name | filter { filter commands } | where filter expression`

Specify a filter to be used for filtering routes coming from the protocol to the routing table. `all` is shorthand for `where true` and `none` is shorthand for `where false`. Default: `all`.

`export filter`

This is similar to the `import` keyword, except that it works in the direction from the routing table to the protocol. Default: `none`.

`import keep filtered switch`

Usually, if an import filter rejects a route, the route is forgotten. When this option is active, these routes are kept in the routing table, but they are hidden and not propagated to other protocols. But it is possible to show them using `show route filtered`. Note that this option does not work for the pipe protocol. Default: `off`.

`import limit [number | off ] [action warn | block | restart | disable]`

Specify an import route limit (a maximum number of routes imported from the protocol) and optionally the action to be taken when the limit is hit. Warn action just prints warning log message. Block action discards new routes coming from the protocol. Restart and disable actions shut the protocol down like appropriate commands. Disable is the default action if an action is not explicitly specified. Note that limits are reset during protocol reconfigure, reload or restart. Default: `off`.

`receive limit [number | off ] [action warn | block | restart | disable]`

Specify an receive route limit (a maximum number of routes received from the protocol and remembered). It works almost identically to `import limit` option, the only difference is that if `import keep filtered` option is active, filtered routes are counted towards the limit and blocked routes are forgotten, as the main purpose of the receive limit is to protect routing tables from overflow. Import limit, on the contrary, counts accepted routes only and routes blocked by the limit are handled like filtered routes. Default: `off`.

`export limit [ number | off ] [action warn | block | restart | disable]`

Specify an export route limit, works similarly to the `import limit` option, but for the routes exported to the protocol. This option is experimental, there are some problems in details of its behavior – the number of exported routes can temporarily exceed the limit without triggering it during protocol reload, exported routes counter ignores route blocking and block action also blocks route updates of already accepted routes – and these details will probably change in the future. Default: `off`.

`description "text"`

This is an optional description of the protocol. It is displayed as a part of the output of `'show protocols all'` command.

`table name`

Connect this protocol to a non-default routing table.

**vrf** *"text"*|**default**

Associate the protocol with specific VRF. The protocol will be restricted to interfaces assigned to the VRF and will use sockets bound to the VRF. A corresponding VRF interface must exist on OS level. For kernel protocol, an appropriate table still must be explicitly selected by **table** option.

By selecting **default**, the protocol is associated with the default VRF; i.e., it will be restricted to interfaces not assigned to any regular VRF. That is different from not specifying **vrf** at all, in which case the protocol may use any interface regardless of its VRF status.

Note that for proper VRF support it is necessary to use Linux kernel version at least 4.14, older versions have limited VRF implementation. Before Linux kernel 5.0, a socket bound to a port in default VRF collide with others in regular VRFs.

There are several options that give sense only with certain protocols:

**interface** [-] [ *"mask"* ] [ *prefix* ] [, ...] [ { *option*; [...] } ]

Specifies a set of interfaces on which the protocol is activated with given interface-specific options. A set of interfaces specified by one interface option is described using an interface pattern. The interface pattern consists of a sequence of clauses (separated by commas), each clause is a mask specified as a shell-like pattern. Interfaces are matched by their name.

An interface matches the pattern if it matches any of its clauses. If the clause begins with -, matching interfaces are excluded. Patterns are processed left-to-right, thus **interface "eth0", -"eth\*", "\*"**; means eth0 and all non-ethernets.

Some protocols (namely OSPFv2 and Direct) support extended clauses that may contain a mask, a prefix, or both of them. An interface matches such clause if its name matches the mask (if specified) and its address matches the prefix (if specified). Extended clauses are used when the protocol handles multiple addresses on an interface independently.

An interface option can be used more times with different interface-specific options, in that case for given interface the first matching interface option is used.

This option is allowed in Babel, BFD, Direct, OSPF, RAdv and RIP protocols, but in OSPF protocol it is used in the **area** subsection.

Default: none.

Examples:

**interface "\*" { type broadcast; };** - start the protocol on all interfaces with **type broadcast** option.

**interface "eth1", "eth4", "eth5" { type ptp; };** - start the protocol on enumerated interfaces with **type ptp** option.

**interface -192.168.1.0/24, 192.168.0.0/16;** - start the protocol on all interfaces that have address from 192.168.0.0/16, but not from 192.168.1.0/24.

**interface -192.168.1.0/24, 192.168.0.0/16;** - start the protocol on all interfaces that have address from 192.168.0.0/16, but not from 192.168.1.0/24.

**interface "eth\*" 192.168.1.0/24;** - start the protocol on all ethernet interfaces that have address from 192.168.1.0/24.

**tx class**|**dscp** *num*

This option specifies the value of ToS/DS/Class field in IP headers of the outgoing protocol packets. This may affect how the protocol packets are processed by the network relative to the other network traffic. With **class** keyword, the value (0-255) is used for the whole ToS/Class octet (but two bits reserved for ECN are ignored). With **dscp** keyword, the value (0-63) is used just for the DS field in the octet. Default value is 0xc0 (DSCP 0x30 - CS6).

**tx priority** *num*

This option specifies the local packet priority. This may affect how the protocol packets are processed in the local TX queues. This option is Linux specific. Default value is 7 (highest priority, privileged traffic).

`password "password" [ { password options } ]`

Specifies a password that can be used by the protocol as a shared secret key. Password option can be used more times to specify more passwords. If more passwords are specified, it is a protocol-dependent decision which one is really used. Specifying passwords does not mean that authentication is enabled, authentication can be enabled by separate, protocol-dependent `authentication` option.

This option is allowed in BFD, OSPF and RIP protocols. BGP has also `password` option, but it is slightly different and described separately. Default: none.

Password option can contain section with some (not necessary all) password sub-options:

`id num`

ID of the password, (1-255). If it is not used, BIRD will choose ID based on an order of the password item in the interface. For example, second password item in one interface will have default ID 2. ID is used by some routing protocols to identify which password was used to authenticate protocol packets.

`generate from "time"`

The start time of the usage of the password for packet signing. The format of *time* is dd-mm-yyyy HH:MM:SS.

`generate to "time"`

The last time of the usage of the password for packet signing.

`accept from "time"`

The start time of the usage of the password for packet verification.

`accept to "time"`

The last time of the usage of the password for packet verification.

`from "time"`

Shorthand for setting both `generate from` and `accept from`.

`to "time"`

Shorthand for setting both `generate to` and `accept to`.

`algorithm ( keyed md5 | keyed sha1 | hmac sha1 | hmac sha256 | hmac sha384 | hmac sha512 )`

The message authentication algorithm for the password when cryptographic authentication is enabled. The default value depends on the protocol. For RIP and OSPFv2 it is Keyed-MD5 (for compatibility), for OSPFv3 protocol it is HMAC-SHA-256.

# Chapter 4: Remote control

You can use the command-line client `birdc` to talk with a running BIRD. Communication is done using a `birdctl` UNIX domain socket (unless changed with the `-s` option given to both the server and the client). The commands can perform simple actions such as enabling/disabling of protocols, telling BIRD to show various information, telling it to show routing table filtered by filter, or asking BIRD to reconfigure. Press `?` at any time to get online help. Option `-r` can be used to enable a restricted mode of BIRD client, which allows just read-only commands (`show ...`). Option `-v` can be passed to the client, to make it dump numeric return codes along with the messages. You do not necessarily need to use `birdc` to talk to BIRD, your own applications could do that, too – the format of communication between BIRD and `birdc` is stable (see the programmer’s documentation).

There is also lightweight variant of BIRD client called `birdcl`, which does not support command line editing and history and has minimal dependencies. This is useful for running BIRD in resource constrained environments, where Readline library (required for regular BIRD client) is not available.

Many commands have the *name* of the protocol instance as an argument. This argument can be omitted if there exists only a single instance.

Here is a brief list of supported functions:

`show status`

Show router status, that is BIRD version, uptime and time from last reconfiguration.

`show interfaces [summary]`

Show the list of interfaces. For each interface, print its type, state, MTU and addresses assigned.

`show protocols [all]`

Show list of protocol instances along with tables they are connected to and protocol status, possibly giving verbose information, if `all` is specified.

`show ospf interface [name] ["interface"]`

Show detailed information about OSPF interfaces.

`show ospf neighbors [name] ["interface"]`

Show a list of OSPF neighbors and a state of adjacency to them.

`show ospf state [all] [name]`

Show detailed information about OSPF areas based on a content of the link-state database. It shows network topology, stub networks, aggregated networks and routers from other areas and external routes. The command shows information about reachable network nodes, use option `all` to show information about all network nodes in the link-state database.

`show ospf topology [all] [name]`

Show a topology of OSPF areas based on a content of the link-state database. It is just a stripped-down version of ‘show ospf state’.

`show ospf lsadb [global | area id | link] [type num] [lsid id] [self | router id] [name]`

Show contents of an OSPF LSA database. Options could be used to filter entries.

`show rip interfaces [name] ["interface"]`

Show detailed information about RIP interfaces.

`show rip neighbors [name] ["interface"]`

Show a list of RIP neighbors and associated state.

`show static [name]`

Show detailed information about static routes.

`show bfd sessions [name]`

Show information about BFD sessions.

`show symbols [table|filter|function|protocol|template|roa|symbol]`

Show the list of symbols defined in the configuration (names of protocols, routing tables etc.).

**show route** [[for] *prefix|IP*] [table *t*] [filter *f*|where *c*] [(export|preexport|noexport) *p*] [protocol *p*] [options]

Show contents of a routing table (by default of the main one or the table attached to a respective protocol), that is routes, their metrics and (in case the **all** switch is given) all their attributes.

You can specify a *prefix* if you want to print routes for a specific network. If you use **for** *prefix or IP*, you'll get the entry which will be used for forwarding of packets to the given destination. By default, all routes for each network are printed with the selected one at the top, unless **primary** is given in which case only the selected route is shown.

You can also ask for printing only routes processed and accepted by a given filter (**filter** *name* or **filter** { *filter* } or matching a given condition (**where** *condition*).

The **export**, **preexport** and **noexport** switches ask for printing of routes that are exported to the specified protocol. With **preexport**, the export filter of the protocol is skipped. With **noexport**, routes rejected by the export filter are printed instead. Note that routes not exported to the protocol for other reasons (e.g. secondary routes or routes imported from that protocol) are not printed even with **noexport**.

You can also select just routes added by a specific protocol. **protocol** *p*.

If BIRD is configured to keep filtered routes (see **import keep filtered** option), you can show them instead of routes by using **filtered** switch.

The **stats** switch requests showing of route statistics (the number of networks, number of routes before and after filtering). If you use **count** instead, only the statistics will be printed.

**mrt dump table** *name|"pattern"* to "*filename*" [filter *f*|where *c*]

Dump content of a routing table to a specified file in MRT table dump format. See [MRT protocol](#) (p. 38) for details.

**show roa** [*prefix* | in *prefix* | for *prefix*] [as *num*] [table *t*]

Show contents of a ROA table (by default of the first one). You can specify a *prefix* to print ROA entries for a specific network. If you use **for** *prefix*, you'll get all entries relevant for route validation of the network prefix; i.e., ROA entries whose prefixes cover the network prefix. Or you can use **in** *prefix* to get ROA entries covered by the network prefix. You could also use **as** option to show just entries for given AS.

**add roa** *prefix* max *num* as *num* [table *t*]

Add a new ROA entry to a ROA table. Such entry is called *dynamic* compared to *static* entries specified in the config file. These dynamic entries survive reconfiguration.

**delete roa** *prefix* max *num* as *num* [table *t*]

Delete the specified ROA entry from a ROA table. Only dynamic ROA entries (i.e., the ones added by **add roa** command) can be deleted.

**flush roa** [table *t*]

Remove all dynamic ROA entries from a ROA table.

**configure** [soft] ["*config file*"] [timeout [num]]

Reload configuration from a given file. BIRD will smoothly switch itself to the new configuration, protocols are reconfigured if possible, restarted otherwise. Changes in filters usually lead to restart of affected protocols.

If **soft** option is used, changes in filters does not cause BIRD to restart affected protocols, therefore already accepted routes (according to old filters) would be still propagated, but new routes would be processed according to the new filters.

If **timeout** option is used, config timer is activated. The new configuration could be either confirmed using **configure confirm** command, or it will be reverted to the old one when the config timer expires. This is useful for cases when reconfiguration breaks current routing and a router becomes inaccessible for an administrator. The config timeout expiration is equivalent to **configure undo** command. The timeout duration could be specified, default is 300 s.

**configure confirm**

Deactivate the config undo timer and therefore confirm the current configuration.

**configure undo**

Undo the last configuration change and smoothly switch back to the previous (stored) configuration. If the last configuration change was soft, the undo change is also soft. There is only one level of undo, but in some specific cases when several reconfiguration requests are given immediately in a row and the intermediate ones are skipped then the undo also skips them back.

**configure check ["*config file*"]**

Read and parse given config file, but do not use it. useful for checking syntactic and some semantic validity of an config file.

**enable|disable|restart *name*|"*pattern*"|all**

Enable, disable or restart a given protocol instance, instances matching the *pattern* or **all** instances.

**reload [*in|out*] *name*|"*pattern*"|all**

Reload a given protocol instance, that means re-import routes from the protocol instance and re-export preferred routes to the instance. If **in** or **out** options are used, the command is restricted to one direction (re-import or re-export).

This command is useful if appropriate filters have changed but the protocol instance was not restarted (or reloaded), therefore it still propagates the old set of routes. For example when **configure soft** command was used to change filters.

Re-export always succeeds, but re-import is protocol-dependent and might fail (for example, if BGP neighbor does not support route-refresh extension). In that case, re-export is also skipped. Note that for the pipe protocol, both directions are always reloaded together (**in** or **out** options are ignored in that case).

**down**

Shut BIRD down.

**debug *protocol|pattern*|all all|off|{ *states|routes|filters|events|packets* [, ...] }**

Control protocol debugging.

**dump *resources|sockets|interfaces|neighbors|attributes|routes|protocols***

Dump contents of internal data structures to the debugging output.

**echo all|off|{ *list of log classes* } [ *buffer-size* ]**

Control echoing of log messages to the command-line output. See [log option](#) (p.7) for a list of log classes.

**eval *expr***

Evaluate given expression.

# Chapter 5: Filters

## 5.1 Introduction

BIRD contains a simple programming language. (No, it can't yet read mail :-). There are two objects in this language: filters and functions. Filters are interpreted by BIRD core when a route is being passed between protocols and routing tables. The filter language contains control structures such as if's and switches, but it allows no loops. An example of a filter using many features can be found in `filter/test.conf`.

Filter gets the route, looks at its attributes and modifies some of them if it wishes. At the end, it decides whether to pass the changed route through (using `accept`) or whether to `reject` it. A simple filter looks like this:

```
filter not_too_far
int var;
{
    if defined( rip_metric ) then
        var = rip_metric;
    else {
        var = 1;
        rip_metric = 1;
    }
    if rip_metric > 10 then
        reject "RIP metric is too big";
    else
        accept "ok";
}
```

As you can see, a filter has a header, a list of local variables, and a body. The header consists of the `filter` keyword followed by a (unique) name of filter. The list of local variables consists of `type name;` pairs where each pair defines one local variable. The body consists of `{ statements }`. Each `statement` is terminated by a `;`. You can group several statements to a single compound statement by using braces (`{ statements }`) which is useful if you want to make a bigger block of code conditional.

BIRD supports functions, so that you don't have to repeat the same blocks of code over and over. Functions can have zero or more parameters and they can have local variables. Recursion is not allowed. Function definitions look like this:

```
function name ()
int local_variable;
{
    local_variable = 5;
}

function with_parameters (int parameter)
{
    print parameter;
}
```

Unlike in C, variables are declared after the `function` line, but before the first `{`. You can't declare variables in nested blocks. Functions are called like in C: `name(); with_parameters(5);`. Function may return values using the `return [expr]` command. Returning a value exits from current function (this is similar to C).

Filters are declared in a way similar to functions except they can't have explicit parameters. They get a route table entry as an implicit parameter, it is also passed automatically to any functions called. The filter must terminate with either `accept` or `reject` statement. If there's a runtime error in filter, the route is rejected.

A nice trick to debug filters is to use `show route filter name` from the command line client. An example session might look like:

```
pavel@bug:~/bird$ ./birdc -s birdctl
BIRD 0.0.0 ready.
bird> show route
10.0.0.0/8          dev eth0 [direct1 23:21] (240)
195.113.30.2/32    dev tunl1 [direct1 23:21] (240)
127.0.0.0/8        dev lo [direct1 23:21] (240)
bird> show route ?
show route [<prefix>] [table <t>] [filter <f>] [all] [primary]...
bird> show route filter { if 127.0.0.5 ~ net then accept; }
127.0.0.0/8        dev lo [direct1 23:21] (240)
bird>
```

## 5.2 Data types

Each variable and each value has certain type. Booleans, integers and enums are incompatible with each other (that is to prevent you from shooting in the foot).

### bool

This is a boolean type, it can have only two values, **true** and **false**. Boolean is the only type you can use in **if** statements.

### int

This is a general integer type. It is an unsigned 32bit type; i.e., you can expect it to store values from 0 to 4294967295. Overflows are not checked. You can use `0x1234` syntax to write hexadecimal values.

### pair

This is a pair of two short integers. Each component can have values from 0 to 65535. Literals of this type are written as `(1234,5678)`. The same syntax can also be used to construct a pair from two arbitrary integer expressions (for example `(1+2,a)`).

### quad

This is a dotted quad of numbers used to represent router IDs (and others). Each component can have a value from 0 to 255. Literals of this type are written like IPv4 addresses.

### string

This is a string of characters. There are no ways to modify strings in filters. You can pass them between functions, assign them to variables of type **string**, print such variables, use standard string comparison operations (e.g. `=`, `!=`, `<`, `>`, `<=`, `>=`), but you can't concatenate two strings. String literals are written as `"This is a string constant"`. Additionally matching (`~`, `!~`) operators could be used to match a string value against a shell pattern (represented also as a string).

### ip

This type can hold a single IP address. Depending on the compile-time configuration of BIRD you are using, it is either an IPv4 or IPv6 address. IP addresses are written in the standard notation (`10.20.30.40` or `fec0:3:4::1`). You can apply special operator `.mask(num)` on values of type **ip**. It masks out all but first `num` bits from the IP address. So `1.2.3.4.mask(8) = 1.0.0.0` is true.

### prefix

This type can hold a network prefix consisting of IP address and prefix length. Prefix literals are written as `ipaddress/palen`, or `ipaddress/netmask`. There are two special operators on prefixes: `.ip` which extracts the IP address from the pair, and `.len`, which separates prefix length from the pair. So `1.2.0.0/16.len = 16` is true.

### ec

This is a specialized type used to represent BGP extended community values. It is essentially a 64bit value, literals of this type are usually written as `(kind, key, value)`, where **kind** is a kind of extended community (e.g. `rt` / `ro` for a route target / route origin communities), the format and possible values of **key** and **value** are usually integers, but it depends on the used kind. Similarly to pairs, ECs can be constructed using expressions for **key** and **value** parts, (e.g. `(ro, myas, 3*10)`, where `myas` is an integer variable).

lc

This is a specialized type used to represent BGP large community values. It is essentially a triplet of 32bit values, where the first value is reserved for the AS number of the issuer, while meaning of remaining parts is defined by the issuer. Literals of this type are written as (123, 456, 789), with any integer values. Similarly to pairs, LCs can be constructed using expressions for its parts, (e.g. (myas, 10+20, 3\*10), where myas is an integer variable).

int|pair|quad|ip|prefix|ec|lc|enum set

Filters recognize four types of sets. Sets are similar to strings: you can pass them around but you can't modify them. Literals of type `int set` look like [ 1, 2, 5..7 ]. As you can see, both simple values and ranges are permitted in sets.

For pair sets, expressions like (123,\*) can be used to denote ranges (in that case (123,0)..(123,65535)). You can also use (123,5..100) for range (123,5)..(123,100). You can also use \* and a..b expressions in the first part of a pair, note that such expressions are translated to a set of intervals, which may be memory intensive. E.g. (\*,4..20) is translated to (0,4..20), (1,4..20), (2,4..20), ... (65535, 4..20).

EC sets use similar expressions like pair sets, e.g. (rt, 123, 10..20) or (ro, 123, \*). Expressions requiring the translation (like (rt, \*, 3)) are not allowed (as they usually have 4B range for ASNs).

Also LC sets use similar expressions like pair sets. You can use ranges and wildcards, but if one field uses that, more specific (later) fields must be wildcards. E.g., (10, 20..30, \*) or (10, 20, 30..40) is valid, while (10, \*, 20..30) or (10, 20..30, 40) is not valid.

You can also use expressions for int, pair, EC and LC set values. However, it must be possible to evaluate these expressions before daemon boots. So you can use only constants inside them. E.g.

```
define one=1;
define myas=64500;
int set odds;
pair set ps;
ec set es;

odds = [ one, 2+1, 6-one, 2*2-1, 9, 11 ];
ps = [ (1,one+one), (3,4)..(4,8), (5,*), (6,3..6), (7..9,*) ];
es = [ (rt, myas, 3*10), (rt, myas+one, 0..16*16*16-1), (ro, myas+2, *) ];
```

Sets of prefixes are special: their literals does not allow ranges, but allows prefix patterns that are written as *ipaddress/pflen{low,high}*. Prefix *ip1/len1* matches prefix pattern *ip2/len2{l,h}* if the first  $\min(\text{len1}, \text{len2})$  bits of *ip1* and *ip2* are identical and  $\text{len1} \leq \text{ip1} \leq \text{len2}$ . A valid prefix pattern has to satisfy  $\text{low} \leq \text{high}$ , but *pflen* is not constrained by *low* or *high*. Obviously, a prefix matches a prefix set literal if it matches any prefix pattern in the prefix set literal.

There are also two shorthands for prefix patterns: *address/len+* is a shorthand for *address/len{len,maxlen}* (where *maxlen* is 32 for IPv4 and 128 for IPv6), that means network prefix *address/len* and all its subnets. *address/len-* is a shorthand for *address/len{0,len}*, that means network prefix *address/len* and all its supernets (network prefixes that contain it).

For example, [ 1.0.0.0/8, 2.0.0.0/8+, 3.0.0.0/8-, 4.0.0.0/8{16,24} ] matches prefix 1.0.0.0/8, all subprefixes of 2.0.0.0/8, all superprefixes of 3.0.0.0/8 and prefixes 4.X.X.X whose prefix length is 16 to 24. [ 0.0.0.0/0{20,24} ] matches all prefixes (regardless of IP address) whose prefix length is 20 to 24, [ 1.2.3.4/32- ] matches any prefix that contains IP address 1.2.3.4. 1.2.0.0/16 ~ [ 1.0.0.0/8{15,17} ] is true, but 1.0.0.0/16 ~ [ 1.0.0.0/8- ] is false.

Cisco-style patterns like 10.0.0.0/8 ge 16 le 24 can be expressed in BIRD as 10.0.0.0/8{16,24}, 192.168.0.0/16 le 24 as 192.168.0.0/16{16,24} and 192.168.0.0/16 ge 24 as 192.168.0.0/16{24,32}.

enum

Enumeration types are fixed sets of possibilities. You can't define your own variables of such type, but some route attributes are of enumeration type. Enumeration types are incompatible with each other.

**bgppath**

BGP path is a list of autonomous system numbers. You can't write literals of this type. There are several special operators on bgppaths:

*P*.**first** returns the first ASN (the neighbor ASN) in path *P*.

*P*.**last** returns the last ASN (the source ASN) in path *P*.

*P*.**last\_nonaggregated** returns the last ASN in the non-aggregated part of the path *P*.

Both **first** and **last** return zero if there is no appropriate ASN, for example if the path contains an AS set element as the first (or the last) part. If the path ends with an AS set, **last\_nonaggregated** may be used to get last ASN before any AS set.

*P*.**len** returns the length of path *P*.

*P*.**empty** resets path *P* to empty path.

**prepend**(*P*, *A*) prepends ASN *A* to path *P* and returns the result.

**delete**(*P*, *A*) deletes all instances of ASN *A* from path *P* and returns the result. *A* may also be an integer set, in that case the operator deletes all ASNs from path *P* that are also members of set *A*.

**filter**(*P*, *A*) deletes all ASNs from path *P* that are not members of integer set *A*. I.e., **filter** do the same as **delete** with inverted set *A*.

Statement *P* = **prepend**(*P*, *A*); can be shortened to *P*.**prepend**(*A*); if *P* is appropriate route attribute (for example **bgp\_path**). Similarly for **delete** and **filter**.

**bgpmask**

BGP masks are patterns used for BGP path matching (using **path** ~ [= 2 3 5 \* =] syntax). The masks resemble wildcard patterns as used by UNIX shells. Autonomous system numbers match themselves, \* matches any (even empty) sequence of arbitrary AS numbers and ? matches one arbitrary AS number. For example, if **bgp\_path** is 4 3 2 1, then: **bgp\_path** ~ [= \* 4 3 \* =] is true, but **bgp\_path** ~ [= \* 4 5 \* =] is false. BGP mask expressions can also contain integer expressions enclosed in parenthesis and integer variables, for example [= \* 4 (1+2) a =]. You can also use ranges, for example [= \* 3..5 2 100..200 \* =]. There is also old (deprecated) syntax that uses / .. / instead of [= .. =] and ? instead of \*.

**clist**

Clist is similar to a set, except that unlike other sets, it can be modified. The type is used for community list (a set of pairs) and for cluster list (a set of quads). There exist no literals of this type. There are three special operators on clists:

*C*.**len** returns the length of clist *C*.

*C*.**empty** resets clist *C* to empty clist.

**add**(*C*, *P*) adds pair (or quad) *P* to clist *C* and returns the result. If item *P* is already in clist *C*, it does nothing. *P* may also be a clist, in that case all its members are added; i.e., it works as clist union.

**delete**(*C*, *P*) deletes pair (or quad) *P* from clist *C* and returns the result. If clist *C* does not contain item *P*, it does nothing. *P* may also be a pair (or quad) set, in that case the operator deletes all items from clist *C* that are also members of set *P*. Moreover, *P* may also be a clist, which works analogously; i.e., it works as clist difference.

**filter**(*C*, *P*) deletes all items from clist *C* that are not members of pair (or quad) set *P*. I.e., **filter** do the same as **delete** with inverted set *P*. *P* may also be a clist, which works analogously; i.e., it works as clist intersection.

Statement *C* = **add**(*C*, *P*); can be shortened to *C*.**add**(*P*); if *C* is appropriate route attribute (for example **bgp\_community**). Similarly for **delete** and **filter**.

**eclist**

Eclist is a data type used for BGP extended community lists. Eclists are very similar to clists, but they are sets of ECs instead of pairs. The same operations (like **add**, **delete** or ~ and !~ membership operators) can be used to modify or test eclists, with ECs instead of pairs as arguments.

**lclist**

Lclist is a data type used for BGP large community lists. Like eclists, lclists are very similar to clists, but they are sets of LCs instead of pairs. The same operations (like `add`, `delete` or `~` and `!~` membership operators) can be used to modify or test lclists, with LCs instead of pairs as arguments.

## 5.3 Operators

The filter language supports common integer operators (`+`, `-`, `*`, `/`), parentheses (`a*(b+c)`), comparison (`a=b`, `a!=b`, `a<b`, `a>=b`). Logical operations include unary not (`!`), and (`&&`) and or (`||`). Special operators include (`~`, `!~`) for "is (not) element of a set" operation - it can be used on element and set of elements of the same type (returning true if element is contained in the given set), or on two strings (returning true if first string matches a shell-like pattern stored in second string) or on IP and prefix (returning true if IP is within the range defined by that prefix), or on prefix and prefix (returning true if first prefix is more specific than second one) or on bgppath and bgpmask (returning true if the path matches the mask) or on number and bgppath (returning true if the number is in the path) or on bgppath and int (number) set (returning true if any ASN from the path is in the set) or on pair/quad and clist (returning true if the pair/quad is element of the clist) or on clist and pair/quad set (returning true if there is an element of the clist that is also a member of the pair/quad set).

There is one operator related to ROA infrastructure - `roa_check()`. It examines a ROA table and does [RFC 6483](#) route origin validation for a given network prefix. The basic usage is `roa_check(table)`, which checks current route (which should be from BGP to have `AS_PATH` argument) in the specified ROA table and returns `ROA_UNKNOWN` if there is no relevant ROA, `ROA_VALID` if there is a matching ROA, or `ROA_INVALID` if there are some relevant ROAs but none of them match. There is also an extended variant `roa_check(table, prefix, asn)`, which allows to specify a prefix and an ASN as arguments.

## 5.4 Control structures

Filters support two control structures: conditions and case switches.

Syntax of a condition is: `if boolean expression then command1; else command2;` and you can use `{ command_1; command_2; ... }` instead of either command. The `else` clause may be omitted. If the *boolean expression* is true, *command1* is executed, otherwise *command2* is executed.

The `case` is similar to case from Pascal. Syntax is `case expr { else: | num_or_prefix [.. num_or_prefix]: statement ; [ ... ] }`. The expression after `case` can be of any type which can be on the left side of the `~` operator and anything that could be a member of a set is allowed before `:`. Multiple commands are allowed without `{}` grouping. If *expr* matches one of the `:` clauses, statements between it and next `:` statement are executed. If *expr* matches neither of the `:` clauses, the statements after `else:` are executed.

Here is example that uses `if` and `case` structures:

```
case arg1 {
    2: print "two";print "I can do more commands without {}";
    3 .. 5: print "three to five";
    else: print "something else";
}

if 1234 = i then printn "."; else {
    print "not 1234";
    print "You need {} around multiple commands";
}
```

## 5.5 Route attributes

A filter is implicitly passed a route, and it can access its attributes just like it accesses variables. Attempts to access undefined attribute result in a runtime error; you can check if an attribute is defined by using

the `defined( attribute )` operator. One notable exception to this rule are attributes of `clist` type, where undefined value is regarded as empty `clist` for most purposes.

*prefix net*

Network the route is talking about. Read-only. (See the chapter about routing tables.)

*enum scope*

The scope of the route. Possible values: `SCOPE_HOST` for routes local to this host, `SCOPE_LINK` for those specific for a physical link, `SCOPE_SITE` and `SCOPE_ORGANIZATION` for private routes and `SCOPE_UNIVERSE` for globally visible routes. This attribute is not interpreted by BIRD and can be used to mark routes in filters. The default value for new routes is `SCOPE_UNIVERSE`.

*int preference*

Preference of the route. Valid values are 0-65535. (See the chapter about routing tables.)

*ip from*

The router which the route has originated from.

*ip gw*

Next hop packets routed using this route should be forwarded to.

*string proto*

The name of the protocol which the route has been imported from. Read-only.

*enum source*

what protocol has told me about this route. Possible values: `RTS_DUMMY`, `RTS_STATIC`, `RTS_INHERIT`, `RTS_DEVICE`, `RTS_STATIC_DEVICE`, `RTS_REDIRECT`, `RTS_RIP`, `RTS_OSPF`, `RTS_OSPF_IA`, `RTS_OSPF_EXT1`, `RTS_OSPF_EXT2`, `RTS_BGP`, `RTS_PIPE`, `RTS_BABEL`.

*enum cast*

Route type (Currently `RTC_UNICAST` for normal routes, `RTC_BROADCAST`, `RTC_MULTICAST`, `RTC_ANYCAST` will be used in the future for broadcast, multicast and anycast routes). Read-only.

*enum dest*

Type of destination the packets should be sent to (`RTD_ROUTER` for forwarding to a neighboring router, `RTD_DEVICE` for routing to a directly-connected network, `RTD_MULTIPATH` for multipath destinations, `RTD_BLACKHOLE` for packets to be silently discarded, `RTD_UNREACHABLE`, `RTD_PROHIBIT` for packets that should be returned with ICMP host unreachable / ICMP administratively prohibited messages). Can be changed, but only to `RTD_BLACKHOLE`, `RTD_UNREACHABLE` or `RTD_PROHIBIT`.

*string ifname*

Name of the outgoing interface. Sink routes (like blackhole, unreachable or prohibit) and multipath routes have no interface associated with them, so `ifname` returns an empty string for such routes. Setting it would also change route to a direct one (remove gateway).

*int ifindex*

Index of the outgoing interface. System wide index of the interface. May be used for interface matching, however indexes might change on interface creation/removal. Zero is returned for routes with undefined outgoing interfaces. Read-only.

*int igp\_metric*

The optional attribute that can be used to specify a distance to the network for routes that do not have a native protocol metric attribute (like `ospf_metric1` for OSPF routes). It is used mainly by BGP to compare internal distances to boundary routers (see below). It is also used when the route is exported to OSPF as a default value for OSPF type 1 metric.

There also exist some protocol-specific attributes which are described in the corresponding protocol sections.

## 5.6 Other statements

The following statements are available:

*variable* = *expr*

Set variable to a given value.

**accept|reject** [ *expr* ]

Accept or reject the route, possibly printing *expr*.

**return** *expr*

Return *expr* from the current function, the function ends at this point.

**print|printn** *expr* [, *expr*...]

Prints given expressions; useful mainly while debugging filters. The **printn** variant does not terminate the line.

**quitbird**

Terminates BIRD. Useful when debugging the filter interpreter.

# Chapter 6: Protocols

## 6.1 Babel

### 6.1.1 Introduction

The Babel protocol ([RFC 6126](#)) is a loop-avoiding distance-vector routing protocol that is robust and efficient both in ordinary wired networks and in wireless mesh networks. Babel is conceptually very simple in its operation and "just works" in its default configuration, though some configuration is possible and in some cases desirable.

While the Babel protocol is dual stack (i.e., can carry both IPv4 and IPv6 routes over the same IPv6 transport), BIRD presently implements only the IPv6 subset of the protocol. No Babel extensions are implemented, but the BIRD implementation can coexist with implementations using the extensions (and will just ignore extension messages).

The Babel protocol implementation in BIRD is currently in alpha stage.

### 6.1.2 Configuration

Babel supports no global configuration options apart from those common to all other protocols, but supports the following per-interface configuration options:

```
protocol babel [<name>] {
    interface <interface pattern> {
        type <wired|wireless>;
        rxcost <number>;
        hello interval <number>;
        update interval <number>;
        port <number>;
        tx class|dscp <number>;
        tx priority <number>;
        rx buffer <number>;
        tx length <number>;
        check link <switch>;
    };
}
```

#### **type** *wired|wireless*

This option specifies the interface type: Wired or wireless. Wired interfaces are considered more reliable, and so the default hello interval is higher, and a neighbour is considered unreachable after only a small number of "hello" packets are lost. On wireless interfaces, hello packets are sent more often, and the ETX link quality estimation technique is used to compute the metrics of routes discovered over this interface. This technique will gradually degrade the metric of routes when packets are lost rather than the more binary up/down mechanism of wired type links. Default: **wired**.

#### **rxcost** *num*

This specifies the RX cost of the interface. The route metrics will be computed from this value with a mechanism determined by the interface **type**. Default: 96 for wired interfaces, 256 for wireless.

#### **hello interval** *num*

Interval at which periodic "hello" messages are sent on this interface, in seconds. Default: 4 seconds.

#### **update interval** *num*

Interval at which periodic (full) updates are sent. Default: 4 times the hello interval.

#### **port** *number*

This option selects an UDP port to operate on. The default is to operate on port 6696 as specified in the Babel RFC.

**tx class|dscp|priority** *number*

These options specify the ToS/DiffServ/Traffic class/Priority of the outgoing Babel packets. See [tx class](#) (p.11) common option for detailed description.

**rx buffer** *number*

This option specifies the size of buffers used for packet processing. The buffer size should be bigger than maximal size of received packets. The default value is the interface MTU, and the value will be clamped to a minimum of 512 bytes + IP packet overhead.

**tx length** *number*

This option specifies the maximum length of generated Babel packets. To avoid IP fragmentation, it should not exceed the interface MTU value. The default value is the interface MTU value, and the value will be clamped to a minimum of 512 bytes + IP packet overhead.

**check link** *switch*

If set, the hardware link state (as reported by OS) is taken into consideration. When the link disappears (e.g. an ethernet cable is unplugged), neighbors are immediately considered unreachable and all routes received from them are withdrawn. It is possible that some hardware drivers or platforms do not implement this feature. Default: yes.

### 6.1.3 Attributes

Babel defines just one attribute: the internal babel metric of the route. It is exposed as the `babel_metric` attribute and has range from 1 to infinity (65535).

### 6.1.4 Example

```
protocol babel {
    interface "eth*" {
        type wired;
    };
    interface "wlan0", "wlan1" {
        type wireless;
        hello interval 1;
        rxcost 512;
    };
    interface "tap0";

    # This matches the default of babeld: redistribute all addresses
    # configured on local interfaces, plus re-distribute all routes received
    # from other babel peers.

    export where (source = RTS_DEVICE) || (source = RTS_BABEL);
}
```

## 6.2 BFD

### 6.2.1 Introduction

Bidirectional Forwarding Detection (BFD) is not a routing protocol itself, it is an independent tool providing liveness and failure detection. Routing protocols like OSPF and BGP use integrated periodic "hello" messages to monitor liveness of neighbors, but detection times of these mechanisms are high (e.g. 40 seconds by default in OSPF, could be set down to several seconds). BFD offers universal, fast and low-overhead mechanism for failure detection, which could be attached to any routing protocol in an advisory role.

BFD consists of mostly independent BFD sessions. Each session monitors an unicast bidirectional path between two BFD-enabled routers. This is done by periodically sending control packets in both directions. BFD does not handle neighbor discovery, BFD sessions are created on demand by request of other protocols

(like OSPF or BGP), which supply appropriate information like IP addresses and associated interfaces. When a session changes its state, these protocols are notified and act accordingly (e.g. break an OSPF adjacency when the BFD session went down).

BIRD implements basic BFD behavior as defined in [RFC 5880](#) (some advanced features like the echo mode or authentication are not implemented), IP transport for BFD as defined in [RFC 5881](#) and [RFC 5883](#) and interaction with client protocols as defined in [RFC 5882](#).

Note that BFD implementation in BIRD is currently a new feature in development, expect some rough edges and possible UI and configuration changes in the future. Also note that we currently support at most one protocol instance.

BFD packets are sent with a dynamic source port number. Linux systems use by default a bit different dynamic port range than the IANA approved one (49152-65535). If you experience problems with compatibility, please adjust `/proc/sys/net/ipv4/ip_local_port_range`.

## 6.2.2 Configuration

BFD configuration consists mainly of multiple definitions of interfaces. Most BFD config options are session specific. When a new session is requested and dynamically created, it is configured from one of these definitions. For sessions to directly connected neighbors, `interface` definitions are chosen based on the interface associated with the session, while `multihop` definition is used for multihop sessions. If no definition is relevant, the session is just created with the default configuration. Therefore, an empty BFD configuration is often sufficient.

Note that to use BFD for other protocols like OSPF or BGP, these protocols also have to be configured to request BFD sessions, usually by `bfd` option.

A BFD instance not associated with any VRF handles session requests from all other protocols, even ones associated with a VRF. Such setup would work for single-hop BFD sessions if `net.ipv4.udp_l3mdev_accept_sysctl` is enabled, but does not currently work for multihop sessions. Another approach is to configure multiple BFD instances, one for each VRF (including the default VRF). Each BFD instance associated with a VRF (regular or default) only handles session requests from protocols in the same VRF.

Some of BFD session options require *time* value, which has to be specified with the appropriate unit: `num s|ms|us`. Although microseconds are allowed as units, practical minimum values are usually in order of tens of milliseconds.

```
protocol bfd [<name>] {
    interface <interface pattern> {
        interval <time>;
        min rx interval <time>;
        min tx interval <time>;
        idle tx interval <time>;
        multiplier <num>;
        passive <switch>;
        authentication none;
        authentication simple;
        authentication [meticulous] keyed md5|sha1;
        password "<text>";
        password "<text>" {
            id <num>;
            generate from "<date>";
            generate to "<date>";
            accept from "<date>";
            accept to "<date>";
            from "<date>";
            to "<date>";
        };
    };
};
multihop {
    interval <time>;
```

```

        min rx interval <time>;
        min tx interval <time>;
        idle tx interval <time>;
        multiplier <num>;
        passive <switch>;
    };
    neighbor <ip> [dev "<interface>"] [local <ip>] [multihop <switch>];
}

```

**interface** *pattern* [, ...] { *options* }

Interface definitions allow to specify options for sessions associated with such interfaces and also may contain interface specific options. See [interface](#) (p.11) common option for a detailed description of interface patterns. Note that contrary to the behavior of **interface** definitions of other protocols, BFD protocol would accept sessions (in default configuration) even on interfaces not covered by such definitions.

**multihop** { *options* }

Multihop definitions allow to specify options for multihop BFD sessions, in the same manner as **interface** definitions are used for directly connected sessions. Currently only one such definition (for all multihop sessions) could be used.

**neighbor** *ip* [dev "*interface*"] [local *ip*] [multihop *switch*]

BFD sessions are usually created on demand as requested by other protocols (like OSPF or BGP). This option allows to explicitly add a BFD session to the specified neighbor regardless of such requests.

The session is identified by the IP address of the neighbor, with optional specification of used interface and local IP. By default the neighbor must be directly connected, unless the session is configured as multihop. Note that local IP must be specified for multihop sessions.

Session specific options (part of **interface** and **multihop** definitions):

**interval** *time*

BFD ensures availability of the forwarding path associated with the session by periodically sending BFD control packets in both directions. The rate of such packets is controlled by two options, **min rx interval** and **min tx interval** (see below). This option is just a shorthand to set both of these options together.

**min rx interval** *time*

This option specifies the minimum RX interval, which is announced to the neighbor and used there to limit the neighbor's rate of generated BFD control packets. Default: 10 ms.

**min tx interval** *time*

This option specifies the desired TX interval, which controls the rate of generated BFD control packets (together with **min rx interval** announced by the neighbor). Note that this value is used only if the BFD session is up, otherwise the value of **idle tx interval** is used instead. Default: 100 ms.

**idle tx interval** *time*

In order to limit unnecessary traffic in cases where a neighbor is not available or not running BFD, the rate of generated BFD control packets is lower when the BFD session is not up. This option specifies the desired TX interval in such cases instead of **min tx interval**. Default: 1 s.

**multiplier** *num*

Failure detection time for BFD sessions is based on established rate of BFD control packets (**min rx/tx interval**) multiplied by this multiplier, which is essentially (ignoring jitter) a number of missed packets after which the session is declared down. Note that rates and multipliers could be different in each direction of a BFD session. Default: 5.

**passive** *switch*

Generally, both BFD session endpoints try to establish the session by sending control packets to the other side. This option allows to enable passive mode, which means that the router does not send BFD packets until it has received one from the other side. Default: disabled.

**authentication none**

No passwords are sent in BFD packets. This is the default value.

**authentication simple**

Every packet carries 16 bytes of password. Received packets lacking this password are ignored. This authentication mechanism is very weak.

**authentication [meticulous] keyed md5|sha1**

An authentication code is appended to each packet. The cryptographic algorithm is keyed MD5 or keyed SHA-1. Note that the algorithm is common for all keys (on one interface), in contrast to OSPF or RIP, where it is a per-key option. Passwords (keys) are not sent open via network.

The **meticulous** variant means that cryptographic sequence numbers are increased for each sent packet, while in the basic variant they are increased about once per second. Generally, the **meticulous** variant offers better resistance to replay attacks but may require more computation.

**password "text"**

Specifies a password used for authentication. See [password](#) (p.12) common option for detailed description. Note that password option **algorithm** is not available in BFD protocol. The algorithm is selected by **authentication** option for all passwords.

**6.2.3 Example**

```
protocol bfd {
    interface "eth*" {
        min rx interval 20 ms;
        min tx interval 50 ms;
        idle tx interval 300 ms;
    };
    interface "gre*" {
        interval 200 ms;
        multiplier 10;
        passive;
    };
    multihop {
        interval 200 ms;
        multiplier 10;
    };

    neighbor 192.168.1.10;
    neighbor 192.168.2.2 dev "eth2";
    neighbor 192.168.10.1 local 192.168.1.1 multihop;
}
```

**6.3 BGP**

The Border Gateway Protocol is the routing protocol used for backbone level routing in the today's Internet. Contrary to other protocols, its convergence does not rely on all routers following the same rules for route selection, making it possible to implement any routing policy at any router in the network, the only restriction being that if a router advertises a route, it must accept and forward packets according to it.

BGP works in terms of autonomous systems (often abbreviated as AS). Each AS is a part of the network with common management and common routing policy. It is identified by a unique 16-bit number (ASN). Routers within each AS usually exchange AS-internal routing information with each other using an interior gateway protocol (IGP, such as OSPF or RIP). Boundary routers at the border of the AS communicate global (inter-AS) network reachability information with their neighbors in the neighboring AS'es via exterior BGP (eBGP) and redistribute received information to other routers in the AS via interior BGP (iBGP).

Each BGP router sends to its neighbors updates of the parts of its routing table it wishes to export along with complete path information (a list of AS'es the packet will travel through if it uses the particular route)

in order to avoid routing loops.

BIRD supports all requirements of the BGP4 standard as defined in RFC 4271. It also supports the community attributes (RFC 1997), capability negotiation (RFC 5492), MD5 password authentication (RFC 2385), extended communities (RFC 4360), route reflectors (RFC 4456), graceful restart (RFC 4724), multiprotocol extensions (RFC 4760), 4B AS numbers (RFC 4893), and 4B AS numbers in extended communities (RFC 5668).

For IPv6, it uses the standard multiprotocol extensions defined in RFC 4760 and applied to IPv6 according to RFC 2545.

### 6.3.1 Route selection rules

BGP doesn't have any simple metric, so the rules for selection of an optimal route among multiple BGP routes with the same preference are a bit more complex and they are implemented according to the following algorithm. It starts the first rule, if there are more "best" routes, then it uses the second rule to choose among them and so on.

- Prefer route with the highest Local Preference attribute.
- Prefer route with the shortest AS path.
- Prefer IGP origin over EGP and EGP origin over incomplete.
- Prefer the lowest value of the Multiple Exit Discriminator.
- Prefer routes received via eBGP over ones received via iBGP.
- Prefer routes with lower internal distance to a boundary router.
- Prefer the route with the lowest value of router ID of the advertising router.

### 6.3.2 IGP routing table

BGP is mainly concerned with global network reachability and with routes to other autonomous systems. When such routes are redistributed to routers in the AS via BGP, they contain IP addresses of a boundary routers (in route attribute NEXT\_HOP). BGP depends on existing IGP routing table with AS-internal routes to determine immediate next hops for routes and to know their internal distances to boundary routers for the purpose of BGP route selection. In BIRD, there is usually one routing table used for both IGP routes and BGP routes.

### 6.3.3 Configuration

Each instance of the BGP corresponds to one neighboring router. This allows to set routing policy and all the other parameters differently for each neighbor using the following configuration parameters:

`local [ip] as number`

Define which AS we are part of. (Note that contrary to other IP routers, BIRD is able to act as a router located in multiple AS'es simultaneously, but in such cases you need to tweak the BGP paths manually in the filters to get consistent behavior.) Optional `ip` argument specifies a source address, equivalent to the `source address` option (see below). This parameter is mandatory.

`neighbor [ip] [port number] [as number]`

Define neighboring router this instance will be talking to and what AS it is located in. In case the neighbor is in the same AS as we are, we automatically switch to iBGP. Optionally, the remote port may also be specified. The parameter may be used multiple times with different sub-options (e.g., both `neighbor 10.0.0.1 as 65000`; and `neighbor 10.0.0.1`; `neighbor as 65000`; are valid). This parameter is mandatory.

**interface** *string*

Define interface we should use for link-local BGP IPv6 sessions. Interface can also be specified as a part of **neighbor address** (e.g., **neighbor fe80::1234%eth0 as 65000**;) . The option may also be used for non link-local sessions when it is necessary to explicitly specify an interface, but only for direct (not multihop) sessions.

**direct**

Specify that the neighbor is directly connected. The IP address of the neighbor must be from a directly reachable IP range (i.e. associated with one of your router's interfaces), otherwise the BGP session wouldn't start but it would wait for such interface to appear. The alternative is the **multihop** option. Default: enabled for eBGP.

**multihop** [*number*]

Configure multihop BGP session to a neighbor that isn't directly connected. Accurately, this option should be used if the configured neighbor IP address does not match with any local network subnets. Such IP address have to be reachable through system routing table. The alternative is the **direct** option. For multihop BGP it is recommended to explicitly configure the source address to have it stable. Optional **number** argument can be used to specify the number of hops (used for TTL). Note that the number of networks (edges) in a path is counted; i.e., if two BGP speakers are separated by one router, the number of hops is 2. Default: enabled for iBGP.

**source address** *ip*

Define local address we should use for next hop calculation and as a source address for the BGP session. Default: the address of the local end of the interface our neighbor is connected to.

**next hop self**

Avoid calculation of the Next Hop attribute and always advertise our own source address as a next hop. This needs to be used only occasionally to circumvent misconfigurations of other routers. Default: disabled.

**next hop keep**

Forward the received Next Hop attribute even in situations where the local address should be used instead, like when the route is sent to an interface with a different subnet. Default: disabled.

**missing lladdr self|drop|ignore**

Next Hop attribute in BGP-IPv6 sometimes contains just the global IPv6 address, but sometimes it has to contain both global and link-local IPv6 addresses. This option specifies what to do if BIRD have to send both addresses but does not know link-local address. This situation might happen when routes from other protocols are exported to BGP, or when improper updates are received from BGP peers. **self** means that BIRD advertises its own local address instead. **drop** means that BIRD skips that prefixes and logs error. **ignore** means that BIRD ignores the problem and sends just the global address (and therefore forms improper BGP update). Default: **self**, unless BIRD is configured as a route server (option **rs client**), in that case default is **ignore**, because route servers usually do not forward packets themselves.

**gateway direct|recursive**

For received routes, their **gw** (immediate next hop) attribute is computed from received **bgp\_next\_hop** attribute. This option specifies how it is computed. Direct mode means that the IP address from **bgp\_next\_hop** is used if it is directly reachable, otherwise the neighbor IP address is used. Recursive mode means that the gateway is computed by an IGP routing table lookup for the IP address from **bgp\_next\_hop**. Note that there is just one level of indirection in recursive mode - the route obtained by the lookup must not be recursive itself, to prevent mutually recursive routes.

Recursive mode is the behavior specified by the BGP standard. Direct mode is simpler, does not require any routes in a routing table, and was used in older versions of BIRD, but does not handle well nontrivial iBGP setups and multihop. Recursive mode is incompatible with [sorted tables](#) (p.6). Default: **direct** for direct sessions, **recursive** for multihop sessions.

**igp table** *name*

Specifies a table that is used as an IGP routing table. Default: the same as the table BGP is connected to.

**check link** *switch*

BGP could use hardware link state into consideration. If enabled, BIRD tracks the link state of the associated interface and when link disappears (e.g. an ethernet cable is unplugged), the BGP session is immediately shut down. Note that this option cannot be used with multihop BGP. Default: disabled.

**bfd** *switch|graceful*

BGP could use BFD protocol as an advisory mechanism for neighbor liveness and failure detection. If enabled, BIRD setups a BFD session for the BGP neighbor and tracks its liveness by it. This has an advantage of an order of magnitude lower detection times in case of failure. When a neighbor failure is detected, the BGP session is restarted. Optionally, it can be configured (by **graceful** argument) to trigger graceful restart instead of regular restart. Note that BFD protocol also has to be configured, see [BFD](#) (p. 24) section for details. Default: disabled.

**ttl security** *switch*

Use GTSM ([RFC 5082](#) - the generalized TTL security mechanism). GTSM protects against spoofed packets by ignoring received packets with a smaller than expected TTL. To work properly, GTSM have to be enabled on both sides of a BGP session. If both **ttl security** and **multihop** options are enabled, **multihop** option should specify proper hop value to compute expected TTL. Kernel support required: Linux: 2.6.34+ (IPv4), 2.6.35+ (IPv6), BSD: since long ago, IPv4 only. Note that full (ICMP protection, for example) [RFC 5082](#) support is provided by Linux only. Default: disabled.

**password** *string*

Use this password for MD5 authentication of BGP sessions ([RFC 2385](#)). When used on BSD systems, see also **setkey** option below. Default: no authentication.

**setkey** *switch*

On BSD systems, keys for TCP MD5 authentication are stored in the global SA/SP database, which can be accessed by external utilities (e.g. `setkey(8)`). BIRD configures security associations in the SA/SP database automatically based on **password** options (see above), this option allows to disable automatic updates by BIRD when manual configuration by external utilities is preferred. Note that automatic SA/SP database updates are currently implemented only for FreeBSD. Passwords have to be set manually by an external utility on NetBSD and OpenBSD. Default: enabled (ignored on non-FreeBSD).

**passive** *switch*

Standard BGP behavior is both initiating outgoing connections and accepting incoming connections. In passive mode, outgoing connections are not initiated. Default: off.

**rr client**

Be a route reflector and treat the neighbor as a route reflection client. Default: disabled.

**rr cluster id** *IPv4 address*

Route reflectors use cluster id to avoid route reflection loops. When there is one route reflector in a cluster it usually uses its router id as a cluster id, but when there are more route reflectors in a cluster, these need to be configured (using this option) to use a common cluster id. Clients in a cluster need not know their cluster id and this option is not allowed for them. Default: the same as router id.

**rs client**

Be a route server and treat the neighbor as a route server client. A route server is used as a replacement for full mesh EBGW routing in Internet exchange points in a similar way to route reflectors used in IBGP routing. BIRD does not implement obsoleted [RFC 1863](#), but uses ad-hoc implementation, which behaves like plain EBGW but reduces modifications to advertised route attributes to be transparent (for example does not prepend its AS number to AS PATH attribute and keeps MED attribute). Default: disabled.

**secondary** *switch*

Usually, if an export filter rejects a selected route, no other route is propagated for that network. This option allows to try the next route in order until one that is accepted is found or all routes for that network are rejected. This can be used for route servers that need to propagate different tables to each client but do not want to have these tables explicitly (to conserve memory). This option requires that the connected routing table is [sorted](#) (p. 6). Default: off.

`add paths switch|rx|tx`

Standard BGP can propagate only one path (route) per destination network (usually the selected one). This option controls the add-path protocol extension, which allows to advertise any number of paths to a destination. Note that to be active, add-path has to be enabled on both sides of the BGP session, but it could be enabled separately for RX and TX direction. When active, all available routes accepted by the export filter are advertised to the neighbor. Default: off.

`allow bgp_local_pref switch`

A standard BGP implementation do not send the Local Preference attribute to eBGP neighbors and ignore this attribute if received from eBGP neighbors, as per [RFC 4271](#). When this option is enabled on an eBGP session, this attribute will be sent to and accepted from the peer, which is useful for example if you have a setup like in [RFC 7938](#). The option does not affect iBGP sessions. Default: off.

`allow local as [number]`

BGP prevents routing loops by rejecting received routes with the local AS number in the AS path. This option allows to loose or disable the check. Optional `number` argument can be used to specify the maximum number of local ASNs in the AS path that is allowed for received routes. When the option is used without the argument, the check is completely disabled and you should ensure loop-free behavior by some other means. Default: 0 (no local AS number allowed).

`enable route refresh switch`

After the initial route exchange, BGP protocol uses incremental updates to keep BGP speakers synchronized. Sometimes (e.g., if BGP speaker changes its import filter, or if there is suspicion of inconsistency) it is necessary to do a new complete route exchange. BGP protocol extension Route Refresh ([RFC 2918](#)) allows BGP speaker to request re-advertisement of all routes from its neighbor. BGP protocol extension Enhanced Route Refresh ([RFC 7313](#)) specifies explicit begin and end for such exchanges, therefore the receiver can remove stale routes that were not advertised during the exchange. This option specifies whether BIRD advertises these capabilities and supports related procedures. Note that even when disabled, BIRD can send route refresh requests. Default: on.

`graceful restart switch|aware`

When a BGP speaker restarts or crashes, neighbors will discard all received paths from the speaker, which disrupts packet forwarding even when the forwarding plane of the speaker remains intact. [RFC 4724](#) specifies an optional graceful restart mechanism to alleviate this issue. This option controls the mechanism. It has three states: Disabled, when no support is provided. Aware, when the graceful restart support is announced and the support for restarting neighbors is provided, but no local graceful restart is allowed (i.e. receiving-only role). Enabled, when the full graceful restart support is provided (i.e. both restarting and receiving role). Note that proper support for local graceful restart requires also configuration of other protocols. Default: aware.

`graceful restart time number`

The restart time is announced in the BGP graceful restart capability and specifies how long the neighbor would wait for the BGP session to re-establish after a restart before deleting stale routes. Default: 120 seconds.

`long lived graceful restart switch|aware`

The long-lived graceful restart is an extension of the traditional [BGP graceful restart](#) (p. 31), where stale routes are kept even after the [restart time](#) (p. 31) expires for additional long-lived stale time, but they are marked with the LLGR\_STALE community, depreferenced, and withdrawn from routers not supporting LLGR. Like traditional BGP graceful restart, it has three states: disabled, aware (receiving-only), and enabled. Note that long-lived graceful restart requires at least aware level of traditional BGP graceful restart. Default: aware, unless graceful restart is disabled.

`long lived stale time number`

The long-lived stale time is announced in the BGP long-lived graceful restart capability and specifies how long the neighbor would keep stale routes depreferenced during long-lived graceful restart until either the session is re-established and synchronized or the stale time expires and routes are removed. Default: 3600 seconds.

`interpret communities switch`

[RFC 1997](#) demands that BGP speaker should process well-known communities like no-export (65535,

65281) or no-advertise (65535, 65282). For example, received route carrying a no-advertise community should not be advertised to any of its neighbors. If this option is enabled (which is by default), BIRD has such behavior automatically (it is evaluated when a route is exported to the BGP protocol just before the export filter). Otherwise, this integrated processing of well-known communities is disabled. In that case, similar behavior can be implemented in the export filter. Default: on.

**enable as4** *switch*

BGP protocol was designed to use 2B AS numbers and was extended later to allow 4B AS number. BIRD supports 4B AS extension, but by disabling this option it can be persuaded not to advertise it and to maintain old-style sessions with its neighbors. This might be useful for circumventing bugs in neighbor's implementation of 4B AS extension. Even when disabled (off), BIRD behaves internally as AS4-aware BGP router. Default: on.

**enable extended messages** *switch*

The BGP protocol uses maximum message length of 4096 bytes. This option provides an extension to allow extended messages with length up to 65535 bytes. Default: off.

**capabilities** *switch*

Use capability advertisement to advertise optional capabilities. This is standard behavior for newer BGP implementations, but there might be some older BGP implementations that reject such connection attempts. When disabled (off), features that request it (4B AS support) are also disabled. Default: on, with automatic fallback to off when received capability-related error.

**advertise ipv4** *switch*

Advertise IPv4 multiprotocol capability. This is not a correct behavior according to the strict interpretation of [RFC 4760](#), but it is widespread and required by some BGP implementations (Cisco and Quagga). This option is relevant to IPv4 mode with enabled capability advertisement only. Default: on.

**route limit** *number*

The maximal number of routes that may be imported from the protocol. If the route limit is exceeded, the connection is closed with an error. Limit is currently implemented as `import limit number action restart`. This option is obsolete and it is replaced by [import limit option](#) (p. 10). Default: no limit.

**disable after error** *switch*

When an error is encountered (either locally or by the other side), disable the instance automatically and wait for an administrator to fix the problem manually. Default: off.

**disable after cease** *switch|set-of-flags*

When a Cease notification is received, disable the instance automatically and wait for an administrator to fix the problem manually. When used with *switch* argument, it means handle every Cease subtype with the exception of `connection collision`. Default: off.

The *set-of-flags* allows to narrow down relevant Cease subtypes. The syntax is `{flag [, ...]}`, where flags are: `cease`, `prefix limit hit`, `administrative shutdown`, `peer deconfigured`, `administrative reset`, `connection rejected`, `configuration change`, `connection collision`, `out of resources`.

**hold time** *number*

Time in seconds to wait for a Keepalive message from the other side before considering the connection stale. Default: depends on agreement with the neighboring router, we prefer 240 seconds if the other side is willing to accept it.

**startup hold time** *number*

Value of the hold timer used before the routers have a chance to exchange open messages and agree on the real value. Default: 240 seconds.

**keepalive time** *number*

Delay in seconds between sending of two consecutive Keepalive messages. Default: One third of the hold time.

`connect delay time` *number*

Delay in seconds between protocol startup and the first attempt to connect. Default: 5 seconds.

`connect retry time` *number*

Time in seconds to wait before retrying a failed attempt to connect. Default: 120 seconds.

`error wait time` *number, number*

Minimum and maximum delay in seconds between a protocol failure (either local or reported by the peer) and automatic restart. Doesn't apply when `disable after error` is configured. If consecutive errors happen, the delay is increased exponentially until it reaches the maximum. Default: 60, 300.

`error forget time` *number*

Maximum time in seconds between two protocol failures to treat them as a error sequence which makes `error wait time` increase exponentially. Default: 300 seconds.

`path metric` *switch*

Enable comparison of path lengths when deciding which BGP route is the best one. Default: on.

`med metric` *switch*

Enable comparison of MED attributes (during best route selection) even between routes received from different ASes. This may be useful if all MED attributes contain some consistent metric, perhaps enforced in import filters of AS boundary routers. If this option is disabled, MED attributes are compared only if routes are received from the same AS (which is the standard behavior). Default: off.

`deterministic med` *switch*

BGP route selection algorithm is often viewed as a comparison between individual routes (e.g. if a new route appears and is better than the current best one, it is chosen as the new best one). But the proper route selection, as specified by [RFC 4271](#), cannot be fully implemented in that way. The problem is mainly in handling the MED attribute. BIRD, by default, uses an simplification based on individual route comparison, which in some cases may lead to temporally dependent behavior (i.e. the selection is dependent on the order in which routes appeared). This option enables a different (and slower) algorithm implementing proper [RFC 4271](#) route selection, which is deterministic. Alternative way how to get deterministic behavior is to use `med metric` option. This option is incompatible with [sorted tables](#) (p.6). Default: off.

`igp metric` *switch*

Enable comparison of internal distances to boundary routers during best route selection. Default: on.

`prefer older` *switch*

Standard route selection algorithm breaks ties by comparing router IDs. This changes the behavior to prefer older routes (when both are external and from different peer). For details, see [RFC 5004](#). Default: off.

`default bgp_med` *number*

Value of the Multiple Exit Discriminator to be used during route selection when the MED attribute is missing. Default: 0.

`default bgp_local_pref` *number*

A default value for the Local Preference attribute. It is used when a new Local Preference attribute is attached to a route by the BGP protocol itself (for example, if a route is received through eBGP and therefore does not have such attribute). Default: 100 (0 in pre-1.2.0 versions of BIRD).

### 6.3.4 Attributes

BGP defines several route attributes. Some of them (those marked with 'I' in the table below) are available on internal BGP connections only, some of them (marked with 'O') are optional.

`bgppath` `bgp_path`

Sequence of AS numbers describing the AS path the packet will travel through when forwarded according to the particular route. In case of internal BGP it doesn't contain the number of the local AS.

`int bgp_local_pref [I]`

Local preference value used for selection among multiple BGP routes (see the selection rules above). It's used as an additional metric which is propagated through the whole local AS.

`int bgp_med [0]`

The Multiple Exit Discriminator of the route is an optional attribute which is used on external (inter-AS) links to convey to an adjacent AS the optimal entry point into the local AS. The received attribute is also propagated over internal BGP links. The attribute value is zeroed when a route is exported to an external BGP instance to ensure that the attribute received from a neighboring AS is not propagated to other neighboring ASes. A new value might be set in the export filter of an external BGP instance. See [RFC 4451](#) for further discussion of BGP MED attribute.

`enum bgp_origin`

Origin of the route: either `ORIGIN_IGP` if the route has originated in an interior routing protocol or `ORIGIN_EGP` if it's been imported from the EGP protocol (nowadays it seems to be obsolete) or `ORIGIN_INCOMPLETE` if the origin is unknown.

`ip bgp_next_hop`

Next hop to be used for forwarding of packets to this destination. On internal BGP connections, it's an address of the originating router if it's inside the local AS or a boundary router the packet will leave the AS through if it's an exterior route, so each BGP speaker within the AS has a chance to use the shortest interior path possible to this point.

`void bgp_atomic_aggr [0]`

This is an optional attribute which carries no value, but the sole presence of which indicates that the route has been aggregated from multiple routes by some router on the path from the originator.

`clist bgp_community [0]`

List of community values associated with the route. Each such value is a pair (represented as a `pair` data type inside the filters) of 16-bit integers, the first of them containing the number of the AS which defines the community and the second one being a per-AS identifier. There are lots of uses of the community mechanism, but generally they are used to carry policy information like "don't export to USA peers". As each AS can define its own routing policy, it also has a complete freedom about which community attributes it defines and what will their semantics be.

`eclist bgp_ext_community [0]`

List of extended community values associated with the route. Extended communities have similar usage as plain communities, but they have an extended range (to allow 4B ASNs) and a nontrivial structure with a type field. Individual community values are represented using an `ec` data type inside the filters.

`lclist bgp_large_community [0]`

List of large community values associated with the route. Large BGP communities is another variant of communities, but contrary to extended communities they behave very much the same way as regular communities, just larger – they are uniform untyped triplets of 32bit numbers. Individual community values are represented using an `lc` data type inside the filters.

`quad bgp_originator_id [I, 0]`

This attribute is created by the route reflector when reflecting the route and contains the router ID of the originator of the route in the local AS.

`clist bgp_cluster_list [I, 0]`

This attribute contains a list of cluster IDs of route reflectors. Each route reflector prepends its cluster ID when reflecting the route.

### 6.3.5 Example

```

protocol bgp {
    local as 65000;                # Use a private AS number
    neighbor 198.51.100.130 as 64496; # Our neighbor ...
    multihop;                      # ... which is connected indirectly
    export filter {                # We use non-trivial export rules
        if source = RTS_STATIC then { # Export only static routes
            # Assign our community
            bgp_community.add((65000,64501));
            # Artificially increase path length
            # by advertising local AS number twice
            if bgp_path ~ [= 65000 =] then
                bgp_path.prepend(65000);
            accept;
        }
        reject;
    };
    import all;
    source address 198.51.100.14; # Use a non-standard source address
}

```

## 6.4 Device

The Device protocol is not a real routing protocol. It doesn't generate any routes and it only serves as a module for getting information about network interfaces from the kernel.

Except for very unusual circumstances, you probably should include this protocol in the configuration since almost all other protocols require network interfaces to be defined for them to work with.

### 6.4.1 Configuration

**scan time** *number*

Time in seconds between two scans of the network interface list. On systems where we are notified about interface status changes asynchronously (such as newer versions of Linux), we need to scan the list only in order to avoid confusion by lost notification messages, so the default time is set to a large value.

**primary** [ "*mask*" ] *prefix*

If a network interface has more than one network address, BIRD has to choose one of them as a primary one. By default, BIRD chooses the lexicographically smallest address as the primary one.

This option allows to specify which network address should be chosen as a primary one. Network addresses that match *prefix* are preferred to non-matching addresses. If more **primary** options are used, the first one has the highest preference. If "*mask*" is specified, then such **primary** option is relevant only to matching network interfaces.

In all cases, an address marked by operating system as secondary cannot be chosen as the primary one.

As the Device protocol doesn't generate any routes, it cannot have any attributes. Example configuration looks like this:

```

protocol device {
    scan time 10;                # Scan the interfaces often
    primary "eth0" 192.168.1.1;
    primary 192.168.0.0/16;
}

```

## 6.5 Direct

The Direct protocol is a simple generator of device routes for all the directly connected networks according to the list of interfaces provided by the kernel via the Device protocol.

The question is whether it is a good idea to have such device routes in BIRD routing table. OS kernel usually handles device routes for directly connected networks by itself so we don't need (and don't want) to export these routes to the kernel protocol. OSPF protocol creates device routes for its interfaces itself and BGP protocol is usually used for exporting aggregate routes. Although there are some use cases that use the direct protocol (like abusing eBGP as an IGP routing protocol), in most cases it is not needed to have these device routes in BIRD routing table and to use the direct protocol.

There is one notable case when you definitely want to use the direct protocol – running BIRD on BSD systems. Having high priority device routes for directly connected networks from the direct protocol protects kernel device routes from being overwritten or removed by IGP routes during some transient network conditions, because a lower priority IGP route for the same network is not exported to the kernel routing table. This is an issue on BSD systems only, as on Linux systems BIRD cannot change non-BIRD route in the kernel routing table.

There are just few configuration options for the Direct protocol:

**interface** *pattern* [, ...]

By default, the Direct protocol will generate device routes for all the interfaces available. If you want to restrict it to some subset of interfaces or addresses (e.g. if you're using multiple routing tables for policy routing and some of the policy domains don't contain all interfaces), just use this clause. See [interface](#) (p. 11) common option for detailed description. The Direct protocol uses extended interface clauses.

**check link** *switch*

If enabled, a hardware link state (reported by OS) is taken into consideration. Routes for directly connected networks are generated only if link up is reported and they are withdrawn when link disappears (e.g., an ethernet cable is unplugged). Default value is no.

Direct device routes don't contain any specific attributes.

Example config might look like this:

```
protocol direct {
    interface "-arc*", "*";           # Exclude the ARCnets
}
```

## 6.6 Kernel

The Kernel protocol is not a real routing protocol. Instead of communicating with other routers in the network, it performs synchronization of BIRD's routing tables with the OS kernel. Basically, it sends all routing table updates to the kernel and from time to time it scans the kernel tables to see whether some routes have disappeared (for example due to unnoticed up/down transition of an interface) or whether an 'alien' route has been added by someone else (depending on the `learn` switch, such routes are either ignored or accepted to our table).

Unfortunately, there is one thing that makes the routing table synchronization a bit more complicated. In the kernel routing table there are also device routes for directly connected networks. These routes are usually managed by OS itself (as a part of IP address configuration) and we don't want to touch that. They are completely ignored during the scan of the kernel tables and also the export of device routes from BIRD tables to kernel routing tables is restricted to prevent accidental interference. This restriction can be disabled using `device routes switch`.

If your OS supports only a single routing table, you can configure only one instance of the Kernel protocol. If it supports multiple tables (in order to allow policy routing; such an OS is for example Linux), you can run as many instances as you want, but each of them must be connected to a different BIRD routing table and to a different kernel table.

Because the kernel protocol is partially integrated with the connected routing table, there are two limitations - it is not possible to connect more kernel protocols to the same routing table and changing route destination

(gateway) in an export filter of a kernel protocol does not work. Both limitations can be overcome using another routing table and the pipe protocol.

### 6.6.1 Configuration

`persist` *switch*

Tell BIRD to leave all its routes in the routing tables when it exits (instead of cleaning them up).

`scan time` *number*

Time in seconds between two consecutive scans of the kernel routing table.

`learn` *switch*

Enable learning of routes added to the kernel routing tables by other routing daemons or by the system administrator. This is possible only on systems which support identification of route authorship.

`device routes` *switch*

Enable export of device routes to the kernel routing table. By default, such routes are rejected (with the exception of explicitly configured device routes from the static protocol) regardless of the export filter to protect device routes in kernel routing table (managed by OS itself) from accidental overwriting or erasing.

`kernel table` *number*

Select which kernel table should this particular instance of the Kernel protocol work with. Available only on systems supporting multiple routing tables.

`metric` *number*

(Linux) Use specified value as a kernel metric (priority) for all routes sent to the kernel. When multiple routes for the same network are in the kernel routing table, the Linux kernel chooses one with lower metric. Also, routes with different metrics do not clash with each other, therefore using dedicated metric value is a reliable way to avoid overwriting routes from other sources (e.g. kernel device routes). Metric 0 has a special meaning of undefined metric, in which either OS default is used, or per-route metric can be set using `krt_metric` attribute. Default: 0 (undefined).

`graceful restart` *switch*

Participate in graceful restart recovery. If this option is enabled and a graceful restart recovery is active, the Kernel protocol will defer synchronization of routing tables until the end of the recovery. Note that import of kernel routes to BIRD is not affected.

`merge paths` *switch* [`limit` *number*]

Usually, only best routes are exported to the kernel protocol. With path merging enabled, both best routes and equivalent non-best routes are merged during export to generate one ECMP (equal-cost multipath) route for each network. This is useful e.g. for BGP multipath. Note that best routes are still pivotal for route export (responsible for most properties of resulting ECMP routes), while exported non-best routes are responsible just for additional multipath next hops. This option also allows to specify a limit on maximal number of nexthops in one route. By default, multipath merging is disabled. If enabled, default value of the limit is 16.

### 6.6.2 Attributes

The Kernel protocol defines several attributes. These attributes are translated to appropriate system (and OS-specific) route attributes. We support these attributes:

`int` `krt_source`

The original source of the imported kernel route. The value is system-dependent. On Linux, it is a value of the protocol field of the route. See `/etc/iproute2/rt_protos` for common values. On BSD, it is based on `STATIC` and `PROTOx` flags. The attribute is read-only.

`int` `krt_metric`

(Linux) The kernel metric of the route. When multiple same routes are in a kernel routing table, the Linux kernel chooses one with lower metric. Note that preferred way to set kernel metric is to use protocol option `metric`, unless per-route metric values are needed.

`ip krt_prefsrc`

(Linux) The preferred source address. Used in source address selection for outgoing packets. Has to be one of the IP addresses of the router.

`int krt_realm`

(Linux) The realm of the route. Can be used for traffic classification.

`int krt_scope`

(Linux IPv4) The scope of the route. Valid values are 0-254, although Linux kernel may reject some values depending on route type and nexthop. It is supposed to represent ‘indirectness’ of the route, where nexthops of routes are resolved through routes with a higher scope, but in current kernels anything below *link* (253) is treated as *global* (0). When not present, global scope is implied for all routes except device routes, where link scope is used by default.

In Linux, there is also a plenty of obscure route attributes mostly focused on tuning TCP performance of local connections. BIRD supports most of these attributes, see Linux or iproute2 documentation for their meaning. Attributes `krt_lock_*` and `krt_feature_*` have type bool, others have type int. Supported attributes are:

`krt_mtu`, `krt_lock_mtu`, `krt_window`, `krt_lock_window`, `krt_rtt`, `krt_lock_rtt`, `krt_rttvar`, `krt_lock_rttvar`, `krt_sstresh`, `krt_lock_sstresh`, `krt_cwnd`, `krt_lock_cwnd`, `krt_advmss`, `krt_lock_advmss`, `krt_reordering`, `krt_lock_reordering`, `krt_hoplimit`, `krt_lock_hoplimit`, `krt_rto_min`, `krt_lock_rto_min`, `krt_initcwnd`, `krt_initrwnd`, `krt_quickack`, `krt_feature_ecn`, `krt_feature_allfrag`

### 6.6.3 Example

A simple configuration can look this way:

```
protocol kernel {
    export all;
}
```

Or for a system with two routing tables:

```
protocol kernel {                                # Primary routing table
    learn;                                       # Learn alien routes from the kernel
    persist;                                    # Don't remove routes on bird shutdown
    scan time 10;                               # Scan kernel routing table every 10 seconds
    import all;
    export all;
}

protocol kernel {                                # Secondary routing table
    table auxetable;
    kernel table 100;
    export all;
}
```

## 6.7 MRT

### 6.7.1 Introduction

The MRT protocol is a component responsible for handling the Multi-Threaded Routing Toolkit (MRT) routing information export format, which is mainly used for collecting and analyzing of routing information from BGP routers. The MRT protocol can be configured to do periodic dumps of routing tables, created MRT files can be analyzed later by other tools. Independent MRT table dumps can also be requested from

BIRD client. There is also a feature to save incoming BGP messages in MRT files, but it is controlled by `mrt_dump` (p. 10) options independently of MRT protocol, although that might change in the future.

BIRD implements the main MRT format specification as defined in [RFC 6396](#) and the ADD\_PATH extension ([RFC 8050](#)).

### 6.7.2 Configuration

MRT configuration consists of several statements describing routing table dumps. Multiple independent periodic dumps can be done as multiple MRT protocol instances. There are two mandatory statements: `filename` and `period`. The behavior can be modified by following configuration parameters:

`table name | "pattern"`

Specify a routing table (or a set of routing tables described by a wildcard pattern) that are to be dumped by the MRT protocol instance. Default: the master table.

`filter { filter commands }`

The MRT protocol allows to specify a filter that is applied to routes as they are dumped. Rejected routes are ignored and not saved to the MRT dump file. Default: no filter.

`where filter expression`

An alternative way to specify a filter for the MRT protocol.

`filename "filename"`

Specify a filename for MRT dump files. The filename may contain time format sequences with *strftime(3)* notation (see *man strftime* for details), there is also a sequence "%N" that is expanded to the name of dumped table. Therefore, each periodic dump of each table can be saved to a different file. Mandatory, see example below.

`period number`

Specify the time interval (in seconds) between periodic dumps. Mandatory.

`always add path switch`

The MRT format uses special records (specified in [RFC 8050](#)) for routes received using BGP ADD\_PATH extension to keep Path ID, while other routes use regular records. This has advantage of better compatibility with tools that do not know special records, but it loses information about which route is the best route. When this option is enabled, both ADD\_PATH and non-ADD\_PATH routes are stored in ADD\_PATH records and order of routes for network is preserved. Default: disabled.

### 6.7.3 Example

```
protocol mrt {
    table "tab*";
    where source = RTS_BGP;
    filename "/var/log/bird/%N_%F_%T.mrt";
    period 300;
}
```

## 6.8 OSPF

### 6.8.1 Introduction

Open Shortest Path First (OSPF) is a quite complex interior gateway protocol. The current IPv4 version (OSPFv2) is defined in [RFC 2328](#) and the current IPv6 version (OSPFv3) is defined in [RFC 5340](#). It's a link state (a.k.a. shortest path first) protocol – each router maintains a database describing the autonomous system's topology. Each participating router has an identical copy of the database and all routers run the

same algorithm calculating a shortest path tree with themselves as a root. OSPF chooses the least cost path as the best path.

In OSPF, the autonomous system can be split to several areas in order to reduce the amount of resources consumed for exchanging the routing information and to protect the other areas from incorrect routing data. Topology of the area is hidden to the rest of the autonomous system.

Another very important feature of OSPF is that it can keep routing information from other protocols (like Static or BGP) in its link state database as external routes. Each external route can be tagged by the advertising router, making it possible to pass additional information between routers on the boundary of the autonomous system.

OSPF quickly detects topological changes in the autonomous system (such as router interface failures) and calculates new loop-free routes after a short period of convergence. Only a minimal amount of routing traffic is involved.

Each router participating in OSPF routing periodically sends Hello messages to all its interfaces. This allows neighbors to be discovered dynamically. Then the neighbors exchange their parts of the link state database and keep it identical by flooding updates. The flooding process is reliable and ensures that each router detects all changes.

## 6.8.2 Configuration

In the main part of configuration, there can be multiple definitions of OSPF areas, each with a different id. These definitions includes many other switches and multiple definitions of interfaces. Definition of interface may contain many switches and constant definitions and list of neighbors on nonbroadcast networks.

```

protocol ospf <name> {
    rfc1583compat <switch>;
    instance id <num>;
    stub router <switch>;
    tick <num>;
    ecmp <switch> [limit <num>];
    merge external <switch>;
    area <id> {
        stub;
        nssa;
        summary <switch>;
        default nssa <switch>;
        default cost <num>;
        default cost2 <num>;
        translator <switch>;
        translator stability <num>;

        networks {
            <prefix>;
            <prefix> hidden;
        }
        external {
            <prefix>;
            <prefix> hidden;
            <prefix> tag <num>;
        }
        stubnet <prefix>;
        stubnet <prefix> {
            hidden <switch>;
            summary <switch>;
            cost <num>;
        }
        interface <interface pattern> [instance <num>] {
            cost <num>;

```

```

    stub <switch>;
    hello <num>;
    poll <num>;
    retransmit <num>;
    priority <num>;
    wait <num>;
    dead count <num>;
    dead <num>;
    secondary <switch>;
    rx buffer [normal|large|<num>];
    tx length <num>;
    type [broadcast|bcast|pointopoint|ptp|
         nonbroadcast|nbma|pointomultipoint|ptmp];
    link lsa suppression <switch>;
    strict nonbroadcast <switch>;
    real broadcast <switch>;
    ptp netmask <switch>;
    check link <switch>;
    bfd <switch>;
    ecmp weight <num>;
    ttl security [<switch>; | tx only]
    tx class|dscp <num>;
    tx priority <num>;
    authentication none|simple|cryptographic;
    password "<text>";
    password "<text>" {
        id <num>;
        generate from "<date>";
        generate to "<date>";
        accept from "<date>";
        accept to "<date>";
        from "<date>";
        to "<date>";
        algorithm ( keyed md5 | keyed sha1 | hmac sha1 | hmac sha25
    };
    neighbors {
        <ip>;
        <ip> eligible;
    };
};
virtual link <id> [instance <num>] {
    hello <num>;
    retransmit <num>;
    wait <num>;
    dead count <num>;
    dead <num>;
    authentication none|simple|cryptographic;
    password "<text>";
    password "<text>" {
        id <num>;
        generate from "<date>";
        generate to "<date>";
        accept from "<date>";
        accept to "<date>";
        from "<date>";
        to "<date>";
        algorithm ( keyed md5 | keyed sha1 | hmac sha1 | hmac sha25

```

```

        };
    };
}

```

**rfc1583compat** *switch*

This option controls compatibility of routing table calculation with [RFC 1583](#). Default value is no.

**instance id** *num*

When multiple OSPF protocol instances are active on the same links, they should use different instance IDs to distinguish their packets. Although it could be done on per-interface basis, it is often preferred to set one instance ID to whole OSPF domain/topology (e.g., when multiple instances are used to represent separate logical topologies on the same physical network). This option specifies the default instance ID for all interfaces of the OSPF instance. Note that this option, if used, must precede interface definitions. Default value is 0.

**stub router** *switch*

This option configures the router to be a stub router, i.e., a router that participates in the OSPF topology but does not allow transit traffic. In OSPFv2, this is implemented by advertising maximum metric for outgoing links. In OSPFv3, the stub router behavior is announced by clearing the R-bit in the router LSA. See [RFC 6987](#) for details. Default value is no.

**tick** *num*

The routing table calculation and clean-up of areas' databases is not performed when a single link state change arrives. To lower the CPU utilization, it's processed later at periodical intervals of *num* seconds. The default value is 1.

**ecmp** *switch* [**limit** *number*]

This option specifies whether OSPF is allowed to generate ECMP (equal-cost multipath) routes. Such routes are used when there are several directions to the destination, each with the same (computed) cost. This option also allows to specify a limit on maximum number of nexthops in one route. By default, ECMP is disabled. If enabled, default value of the limit is 16.

**merge external** *switch*

This option specifies whether OSPF should merge external routes from different routers/LSAs for the same destination. When enabled together with **ecmp**, equal-cost external routes will be combined to multipath routes in the same way as regular routes. When disabled, external routes from different LSAs are treated as separate even if they represents the same destination. Default value is no.

**area** *id*

This defines an OSPF area with given area ID (an integer or an IPv4 address, similarly to a router ID). The most important area is the backbone (ID 0) to which every other area must be connected.

**stub**

This option configures the area to be a stub area. External routes are not flooded into stub areas. Also summary LSAs can be limited in stub areas (see option **summary**). By default, the area is not a stub area.

**nssa**

This option configures the area to be a NSSA (Not-So-Stubby Area). NSSA is a variant of a stub area which allows a limited way of external route propagation. Global external routes are not propagated into a NSSA, but an external route can be imported into NSSA as a (area-wide) NSSA-LSA (and possibly translated and/or aggregated on area boundary). By default, the area is not NSSA.

**summary** *switch*

This option controls propagation of summary LSAs into stub or NSSA areas. If enabled, summary LSAs are propagated as usual, otherwise just the default summary route (0.0.0.0/0) is propagated (this is sometimes called totally stubby area). If a stub area has more area boundary routers, propagating summary LSAs could lead to more efficient routing at the cost of larger link state database. Default value is no.

**default nssa switch**

When **summary** option is enabled, default summary route is no longer propagated to the NSSA. In that case, this option allows to originate default route as NSSA-LSA to the NSSA. Default value is no.

**default cost num**

This option controls the cost of a default route propagated to stub and NSSA areas. Default value is 1000.

**default cost2 num**

When a default route is originated as NSSA-LSA, its cost can use either type 1 or type 2 metric. This option allows to specify the cost of a default route in type 2 metric. By default, type 1 metric (option **default cost**) is used.

**translator switch**

This option controls translation of NSSA-LSAs into external LSAs. By default, one translator per NSSA is automatically elected from area boundary routers. If enabled, this area boundary router would unconditionally translate all NSSA-LSAs regardless of translator election. Default value is no.

**translator stability num**

This option controls the translator stability interval (in seconds). When the new translator is elected, the old one keeps translating until the interval is over. Default value is 40.

**networks { set }**

Definition of area IP ranges. This is used in summary LSA origination. Hidden networks are not propagated into other areas.

**external { set }**

Definition of external area IP ranges for NSSAs. This is used for NSSA-LSA translation. Hidden networks are not translated into external LSAs. Networks can have configured route tag.

**stubnet prefix { options }**

Stub networks are networks that are not transit networks between OSPF routers. They are also propagated through an OSPF area as a part of a link state database. By default, BIRD generates a stub network record for each primary network address on each OSPF interface that does not have any OSPF neighbors, and also for each non-primary network address on each OSPF interface. This option allows to alter a set of stub networks propagated by this router.

Each instance of this option adds a stub network with given network prefix to the set of propagated stub network, unless option **hidden** is used. It also suppresses default stub networks for given network prefix. When option **summary** is used, also default stub networks that are subnetworks of given stub network are suppressed. This might be used, for example, to aggregate generated stub networks.

**interface pattern [instance num]**

Defines that the specified interfaces belong to the area being defined. See [interface](#) (p.11) common option for detailed description. In OSPFv2, extended interface clauses are used, because each network prefix is handled as a separate virtual interface.

You can specify alternative instance ID for the interface definition, therefore it is possible to have several instances of that interface with different options or even in different areas. For OSPFv2, instance ID support is an extension ([RFC 6549](#)) and is supposed to be set per-protocol. For OSPFv3, it is an integral feature.

**virtual link id [instance num]**

Virtual link to router with the router id. Virtual link acts as a point-to-point interface belonging to backbone. The actual area is used as a transport area. This item cannot be in the backbone. Like with **interface** option, you could also use several virtual links to one destination with different instance IDs.

**cost num**

Specifies output cost (metric) of an interface. Default value is 10.

**stub switch**

If set to interface it does not listen to any packet and does not send any hello. Default value is no.

**hello** *num*

Specifies interval in seconds between sending of Hello messages. Beware, all routers on the same network need to have the same hello interval. Default value is 10.

**poll** *num*

Specifies interval in seconds between sending of Hello messages for some neighbors on NBMA network. Default value is 20.

**retransmit** *num*

Specifies interval in seconds between retransmissions of unacknowledged updates. Default value is 5.

**transmit delay** *num*

Specifies estimated transmission delay of link state updates send over the interface. The value is added to LSA age of LSAs propagated through it. Default value is 1.

**priority** *num*

On every multiple access network (e.g., the Ethernet) Designated Router and Backup Designated router are elected. These routers have some special functions in the flooding process. Higher priority increases preferences in this election. Routers with priority 0 are not eligible. Default value is 1.

**wait** *num*

After start, router waits for the specified number of seconds between starting election and building adjacency. Default value is  $4*hello$ .

**dead count** *num*

When the router does not receive any messages from a neighbor in  $dead\ count*hello$  seconds, it will consider the neighbor down.

**dead** *num*

When the router does not receive any messages from a neighbor in  $dead$  seconds, it will consider the neighbor down. If both directives **dead count** and **dead** are used, **dead** has precedence.

**secondary** *switch*

On BSD systems, older versions of BIRD supported OSPFv2 only for the primary IP address of an interface, other IP ranges on the interface were handled as stub networks. Since v1.4.1, regular operation on secondary IP addresses is supported, but disabled by default for compatibility. This option allows to enable it. The option is a transitional measure, will be removed in the next major release as the behavior will be changed. On Linux systems, the option is irrelevant, as operation on non-primary addresses is already the regular behavior.

**rx buffer** *num*

This option allows to specify the size of buffers used for packet processing. The buffer size should be bigger than maximal size of any packets. By default, buffers are dynamically resized as needed, but a fixed value could be specified. Value **large** means maximal allowed packet size - 65535.

**tx length** *num*

Transmitted OSPF messages that contain large amount of information are segmented to separate OSPF packets to avoid IP fragmentation. This option specifies the soft ceiling for the length of generated OSPF packets. Default value is the MTU of the network interface. Note that larger OSPF packets may still be generated if underlying OSPF messages cannot be splitted (e.g. when one large LSA is propagated).

**type** **broadcast|bcast**

BIRD detects a type of a connected network automatically, but sometimes it's convenient to force use of a different type manually. On broadcast networks (like ethernet), flooding and Hello messages are sent using multicasts (a single packet for all the neighbors). A designated router is elected and it is responsible for synchronizing the link-state databases and originating network LSAs. This network type cannot be used on physically NBMA networks and on unnumbered networks (networks without proper IP prefix).

**type pointopoint|ptp**

Point-to-point networks connect just 2 routers together. No election is performed and no network LSA is originated, which makes it simpler and faster to establish. This network type is useful not only for physically PtP ifaces (like PPP or tunnels), but also for broadcast networks used as PtP links. This network type cannot be used on physically NBMA networks.

**type nonbroadcast|nbma**

On NBMA networks, the packets are sent to each neighbor separately because of lack of multicast capabilities. Like on broadcast networks, a designated router is elected, which plays a central role in propagation of LSAs. This network type cannot be used on unnumbered networks.

**type pointomultipoint|ptmp**

This is another network type designed to handle NBMA networks. In this case the NBMA network is treated as a collection of PtP links. This is useful if not every pair of routers on the NBMA network has direct communication, or if the NBMA network is used as an (possibly unnumbered) PtP link.

**link lsa suppression *switch***

In OSPFv3, link LSAs are generated for each link, announcing link-local IPv6 address of the router to its local neighbors. These are useless on PtP or PtMP networks and this option allows to suppress the link LSA origination for such interfaces. The option is ignored on other than PtP or PtMP interfaces. Default value is no.

**strict nonbroadcast *switch***

If set, don't send hello to any undefined neighbor. This switch is ignored on other than NBMA or PtMP interfaces. Default value is no.

**real broadcast *switch***

In **type broadcast** or **type ptp** network configuration, OSPF packets are sent as IP multicast packets. This option changes the behavior to using old-fashioned IP broadcast packets. This may be useful as a workaround if IP multicast for some reason does not work or does not work reliably. This is a non-standard option and probably is not interoperable with other OSPF implementations. Default value is no.

**ptp netmask *switch***

In **type ptp** network configurations, OSPFv2 implementations should ignore received netmask field in hello packets and should send hello packets with zero netmask field on unnumbered PtP links. But some OSPFv2 implementations perform netmask checking even for PtP links. This option specifies whether real netmask will be used in hello packets on **type ptp** interfaces. You should ignore this option unless you meet some compatibility problems related to this issue. Default value is no for unnumbered PtP links, yes otherwise.

**check link *switch***

If set, a hardware link state (reported by OS) is taken into consideration. When a link disappears (e.g. an ethernet cable is unplugged), neighbors are immediately considered unreachable and only the address of the iface (instead of whole network prefix) is propagated. It is possible that some hardware drivers or platforms do not implement this feature. Default value is no.

**bfd *switch***

OSPF could use BFD protocol as an advisory mechanism for neighbor liveness and failure detection. If enabled, BIRD setups a BFD session for each OSPF neighbor and tracks its liveness by it. This has an advantage of an order of magnitude lower detection times in case of failure. Note that BFD protocol also has to be configured, see [BFD](#) (p.24) section for details. Default value is no.

**ttl security [*switch* | tx only]**

TTL security is a feature that protects routing protocols from remote spoofed packets by using TTL 255 instead of TTL 1 for protocol packets destined to neighbors. Because TTL is decremented when packets are forwarded, it is non-trivial to spoof packets with TTL 255 from remote locations. Note that this option would interfere with OSPF virtual links.

If this option is enabled, the router will send OSPF packets with TTL 255 and drop received packets with TTL less than 255. If this option is set to **tx only**, TTL 255 is used for sent packets, but is not checked for received packets. Default value is no.

`tx class|dscp|priority num`

These options specify the ToS/DiffServ/Traffic class/Priority of the outgoing OSPF packets. See [tx class](#) (p. 11) common option for detailed description.

`ecmp weight num`

When ECMP (multipath) routes are allowed, this value specifies a relative weight used for nexthops going through the iface. Allowed values are 1-256. Default value is 1.

`authentication none`

No passwords are sent in OSPF packets. This is the default value.

`authentication simple`

Every packet carries 8 bytes of password. Received packets lacking this password are ignored. This authentication mechanism is very weak. This option is not available in OSPFv3.

`authentication cryptographic`

An authentication code is appended to every packet. The specific cryptographic algorithm is selected by option `algorithm` for each key. The default cryptographic algorithm for OSPFv2 keys is Keyed-MD5 and for OSPFv3 keys is HMAC-SHA-256. Passwords are not sent open via network, so this mechanism is quite secure. Packets can still be read by an attacker.

`password "text"`

Specifies a password used for authentication. See [password](#) (p. 12) common option for detailed description.

`neighbors { set }`

A set of neighbors to which Hello messages on NBMA or PtMP networks are to be sent. For NBMA networks, some of them could be marked as eligible. In OSPFv3, link-local addresses should be used, using global ones is possible, but it is nonstandard and might be problematic. And definitely, link-local and global addresses should not be mixed.

### 6.8.3 Attributes

OSPF defines four route attributes. Each internal route has a `metric`.

Metric is ranging from 1 to infinity (65535). External routes use `metric type 1` or `metric type 2`. A `metric of type 1` is comparable with internal `metric`, a `metric of type 2` is always longer than any `metric of type 1` or any internal `metric`. Internal `metric` or `metric of type 1` is stored in attribute `ospf_metric1`, `metric type 2` is stored in attribute `ospf_metric2`. If you specify both metrics only `metric1` is used.

Each external route can also carry attribute `ospf_tag` which is a 32-bit integer which is used when exporting routes to other protocols; otherwise, it doesn't affect routing inside the OSPF domain at all. The fourth attribute `ospf_router_id` is a router ID of the router advertising that route / network. This attribute is read-only. Default is `ospf_metric2 = 10000` and `ospf_tag = 0`.

### 6.8.4 Example

```
protocol ospf MyOSPF {
    rfc1583compat yes;
    tick 2;
    export filter {
        if source = RTS_BGP then {
            ospf_metric1 = 100;
            accept;
        }
        reject;
    };
    area 0.0.0.0 {
        interface "eth*" {
            cost 11;
        }
    }
}
```

```

        hello 15;
        priority 100;
        retransmit 7;
        authentication simple;
        password "aaa";
    };
interface "ppp*" {
    cost 100;
    authentication cryptographic;
    password "abc" {
        id 1;
        generate to "22-04-2003 11:00:06";
        accept from "17-01-2001 12:01:05";
        algorithm hmac sha384;
    };
    password "def" {
        id 2;
        generate to "22-07-2005 17:03:21";
        accept from "22-02-2001 11:34:06";
        algorithm hmac sha512;
    };
};
interface "arc0" {
    cost 10;
    stub yes;
};
interface "arc1";
};
area 120 {
    stub yes;
    networks {
        172.16.1.0/24;
        172.16.2.0/24 hidden;
    }
    interface "-arc0" , "arc*" {
        type nonbroadcast;
        authentication none;
        strict nonbroadcast yes;
        wait 120;
        poll 40;
        dead count 8;
        neighbors {
            192.168.120.1 eligible;
            192.168.120.2;
            192.168.120.10;
        };
    };
};
}

```

## 6.9 Pipe

### 6.9.1 Introduction

The Pipe protocol serves as a link between two routing tables, allowing routes to be passed from a table declared as primary (i.e., the one the pipe is connected to using the `table` configuration keyword) to the secondary one (declared using `peer table`) and vice versa, depending on what's allowed by the filters.

Export filters control export of routes from the primary table to the secondary one, import filters control the opposite direction.

The Pipe protocol may work in the transparent mode or in the opaque mode. In the transparent mode, the Pipe protocol retransmits all routes from one table to the other table, retaining their original source and attributes. If import and export filters are set to accept, then both tables would have the same content. The transparent mode is the default mode.

In the opaque mode, the Pipe protocol retransmits optimal route from one table to the other table in a similar way like other protocols send and receive routes. Retransmitted route will have the source set to the Pipe protocol, which may limit access to protocol specific route attributes. This mode is mainly for compatibility, it is not suggested for new configs. The mode can be changed by `mode` option.

The primary use of multiple routing tables and the Pipe protocol is for policy routing, where handling of a single packet doesn't depend only on its destination address, but also on its source address, source interface, protocol type and other similar parameters. In many systems (Linux being a good example), the kernel allows to enforce routing policies by defining routing rules which choose one of several routing tables to be used for a packet according to its parameters. Setting of these rules is outside the scope of BIRD's work (on Linux, you can use the `ip` command), but you can create several routing tables in BIRD, connect them to the kernel ones, use filters to control which routes appear in which tables and also you can employ the Pipe protocol for exporting a selected subset of one table to another one.

## 6.9.2 Configuration

`peer table table`

Defines secondary routing table to connect to. The primary one is selected by the `table` keyword.

`mode opaque|transparent`

Specifies the mode for the pipe to work in. Default is transparent.

## 6.9.3 Attributes

The Pipe protocol doesn't define any route attributes.

## 6.9.4 Example

Let's consider a router which serves as a boundary router of two different autonomous systems, each of them connected to a subset of interfaces of the router, having its own exterior connectivity and wishing to use the other AS as a backup connectivity in case of outage of its own exterior line.

Probably the simplest solution to this situation is to use two routing tables (we'll call them `as1` and `as2`) and set up kernel routing rules, so that packets having arrived from interfaces belonging to the first AS will be routed according to `as1` and similarly for the second AS. Thus we have split our router to two logical routers, each one acting on its own routing table, having its own routing protocols on its own interfaces. In order to use the other AS's routes for backup purposes, we can pass the routes between the tables through a Pipe protocol while decreasing their preferences and correcting their BGP paths to reflect the AS boundary crossing.

```

table as1;                               # Define the tables
table as2;

protocol kernel kern1 {                  # Synchronize them with the kernel
    table as1;
    kernel table 1;
}

protocol kernel kern2 {
    table as2;
    kernel table 2;
}

```

```

protocol bgp bgp1 {                                # The outside connections
    table as1;
    local as 1;
    neighbor 192.168.0.1 as 1001;
    export all;
    import all;
}

protocol bgp bgp2 {
    table as2;
    local as 2;
    neighbor 10.0.0.1 as 1002;
    export all;
    import all;
}

protocol pipe {                                    # The Pipe
    table as1;
    peer table as2;
    export filter {
        if net ~ [ 1.0.0.0/8+] then { # Only AS1 networks
            if preference>10 then preference = preference-10;
            if source=RTS_BGP then bgp_path.prepend(1);
            accept;
        }
        reject;
    };
    import filter {
        if net ~ [ 2.0.0.0/8+] then { # Only AS2 networks
            if preference>10 then preference = preference-10;
            if source=RTS_BGP then bgp_path.prepend(2);
            accept;
        }
        reject;
    };
}

```

## 6.10 RAdv

### 6.10.1 Introduction

The RAdv protocol is an implementation of Router Advertisements, which are used in the IPv6 stateless autoconfiguration. IPv6 routers send (in irregular time intervals or as an answer to a request) advertisement packets to connected networks. These packets contain basic information about a local network (e.g. a list of network prefixes), which allows network hosts to autoconfigure network addresses and choose a default route. BIRD implements router behavior as defined in [RFC 4861](#), router preferences and specific routes ([RFC 4191](#)), and DNS extensions ([RFC 6106](#)).

### 6.10.2 Configuration

There are several classes of definitions in RAdv configuration – interface definitions, prefix definitions and DNS definitions:

```
interface pattern [, ...] { options }
```

Interface definitions specify a set of interfaces on which the protocol is activated and contain interface specific options. See [interface](#) (p.11) common options for detailed description.

**prefix** *prefix* { *options* }

Prefix definitions allow to modify a list of advertised prefixes. By default, the advertised prefixes are the same as the network prefixes assigned to the interface. For each network prefix, the matching prefix definition is found and its options are used. If no matching prefix definition is found, the prefix is used with default options.

Prefix definitions can be either global or interface-specific. The second ones are part of interface options. The prefix definition matching is done in the first-match style, when interface-specific definitions are processed before global definitions. As expected, the prefix definition is matching if the network prefix is a subnet of the prefix in prefix definition.

**rdnss** { *options* }

RDNSS definitions allow to specify a list of advertised recursive DNS servers together with their options. As options are seldom necessary, there is also a short variant **rdnss** *address* that just specifies one DNS server. Multiple definitions are cumulative. RDNSS definitions may also be interface-specific when used inside interface options. By default, interface uses both global and interface-specific options, but that can be changed by **rdnss local** option.

**dnssl** { *options* }

DNSSL definitions allow to specify a list of advertised DNS search domains together with their options. Like **rdnss** above, multiple definitions are cumulative, they can be used also as interface-specific options and there is a short variant **dnssl domain** that just specifies one DNS search domain.

**trigger** *prefix*

RAdv protocol could be configured to change its behavior based on availability of routes. When this option is used, the protocol waits in suppressed state until a *trigger route* (for the specified network) is exported to the protocol, the protocol also returns to suppressed state if the *trigger route* disappears. Note that route export depends on specified export filter, as usual. This option could be used, e.g., for handling failover in multihoming scenarios.

During suppressed state, router advertisements are generated, but with some fields zeroed. Exact behavior depends on which fields are zeroed, this can be configured by **sensitive** option for appropriate fields. By default, just **default lifetime** (also called **router lifetime**) is zeroed, which means hosts cannot use the router as a default router. **preferred lifetime** and **valid lifetime** could also be configured as **sensitive** for a prefix, which would cause autoconfigured IPs to be deprecated or even removed.

**propagate routes** *switch*

This option controls propagation of more specific routes, as defined in [RFC 4191](#). If enabled, all routes exported to the RAdv protocol, with the exception of the trigger prefix, are added to advertisements as additional options. The lifetime and preference of advertised routes can be set individually by **ra\_lifetime** and **ra\_preference** route attributes, or per interface by **route lifetime** and **route preference** options. Default: disabled.

Note that the RFC discourages from sending more than 17 routes and recommends the routes to be configured manually.

Interface specific options:

**max ra interval** *expr*

Unsolicited router advertisements are sent in irregular time intervals. This option specifies the maximum length of these intervals, in seconds. Valid values are 4-1800. Default: 600

**min ra interval** *expr*

This option specifies the minimum length of that intervals, in seconds. Must be at least 3 and at most  $3/4 * \text{max ra interval}$ . Default: about  $1/3 * \text{max ra interval}$ .

**min delay** *expr*

The minimum delay between two consecutive router advertisements, in seconds. Default: 3

**managed** *switch*

This option specifies whether hosts should use DHCPv6 for IP address configuration. Default: no

**other config** *switch*

This option specifies whether hosts should use DHCPv6 to receive other configuration information.  
Default: no

**link mtu** *expr*

This option specifies which value of MTU should be used by hosts. 0 means unspecified. Default: 0

**reachable time** *expr*

This option specifies the time (in milliseconds) how long hosts should assume a neighbor is reachable (from the last confirmation). Maximum is 3600000, 0 means unspecified. Default 0.

**retrans timer** *expr*

This option specifies the time (in milliseconds) how long hosts should wait before retransmitting Neighbor Solicitation messages. 0 means unspecified. Default 0.

**current hop limit** *expr*

This option specifies which value of Hop Limit should be used by hosts. Valid values are 0-255, 0 means unspecified. Default: 64

**default lifetime** *expr* [**sensitive** *switch*]

This option specifies the time (in seconds) how long (since the receipt of RA) hosts may use the router as a default router. 0 means do not use as a default router. For **sensitive** option, see [trigger](#) (p. 50).  
Default: 3 \* max ra interval, sensitive yes.

**default preference** low|medium|high

This option specifies the Default Router Preference value to advertise to hosts. Default: medium.

**route lifetime** *expr* [**sensitive** *switch*]

This option specifies the default value of advertised lifetime for specific routes; i.e., the time (in seconds) for how long (since the receipt of RA) hosts should consider these routes valid. A special value 0xffffffff represents infinity. The lifetime can be overridden on a per route basis by the [ra\\_lifetime](#) (p. 52) route attribute. Default: 3 \* max ra interval, sensitive no.

For the **sensitive** option, see [trigger](#) (p. 50). If **sensitive** is enabled, even the routes with the [ra\\_lifetime](#) attribute become sensitive to the trigger.

**route preference** low|medium|high

This option specifies the default value of advertised route preference for specific routes. The value can be overridden on a per route basis by the [ra\\_preference](#) (p. 52) route attribute. Default: medium.

**prefix linger time** *expr*

When a prefix or a route disappears, it is advertised for some time with zero lifetime, to inform clients it is no longer valid. This option specifies the time (in seconds) for how long prefixes are advertised that way. Default: 3 \* max ra interval.

**route linger time** *expr*

When a prefix or a route disappears, it is advertised for some time with zero lifetime, to inform clients it is no longer valid. This option specifies the time (in seconds) for how long routes are advertised that way. Default: 3 \* max ra interval.

**rdnss local** *switch*

Use only local (interface-specific) RDNSS definitions for this interface. Otherwise, both global and local definitions are used. Could also be used to disable RDNSS for given interface if no local definitions are specified. Default: no.

**dnssl local** *switch*

Use only local DNSSL definitions for this interface. See **rdnss local** option above. Default: no.

## Prefix specific options

**skip** *switch*

This option allows to specify that given prefix should not be advertised. This is useful for making exceptions from a default policy of advertising all prefixes. Note that for withdrawing an already advertised prefix it is more useful to advertise it with zero valid lifetime. Default: no

**onlink** *switch*

This option specifies whether hosts may use the advertised prefix for onlink determination. Default: yes

**autonomous** *switch*

This option specifies whether hosts may use the advertised prefix for stateless autoconfiguration. Default: yes

**valid lifetime** *expr* [**sensitive** *switch*]

This option specifies the time (in seconds) how long (after the receipt of RA) the prefix information is valid, i.e., autoconfigured IP addresses can be assigned and hosts with that IP addresses are considered directly reachable. 0 means the prefix is no longer valid. For **sensitive** option, see [trigger](#) (p.50). Default: 86400 (1 day), **sensitive** no.

**preferred lifetime** *expr* [**sensitive** *switch*]

This option specifies the time (in seconds) how long (after the receipt of RA) IP addresses generated from the prefix using stateless autoconfiguration remain preferred. For **sensitive** option, see [trigger](#) (p.50). Default: 14400 (4 hours), **sensitive** no.

RDNSS specific options:

**ns** *address*

This option specifies one recursive DNS server. Can be used multiple times for multiple servers. It is mandatory to have at least one **ns** option in **rdnss** definition.

**lifetime** [**mult**] *expr*

This option specifies the time how long the RDNSS information may be used by clients after the receipt of RA. It is expressed either in seconds or (when **mult** is used) in multiples of **max ra interval**. Note that RDNSS information is also invalidated when **default lifetime** expires. 0 means these addresses are no longer valid DNS servers. Default: 3 \* **max ra interval**.

DNSSL specific options:

**domain** *address*

This option specifies one DNS search domain. Can be used multiple times for multiple domains. It is mandatory to have at least one **domain** option in **dnssl** definition.

**lifetime** [**mult**] *expr*

This option specifies the time how long the DNSSL information may be used by clients after the receipt of RA. Details are the same as for RDNSS **lifetime** option above. Default: 3 \* **max ra interval**.

### 6.10.3 Attributes

RAdv defines two route attributes:

**enum** *ra\_preference*

The preference of the route. The value can be *RA\_PREF\_LOW*, *RA\_PREF\_MEDIUM* or *RA\_PREF\_HIGH*. If the attribute is not set, the [route preference](#) (p.51) option is used.

**int** *ra\_lifetime*

The advertised lifetime of the route, in seconds. The special value of 0xffffffff represents infinity. If the attribute is not set, the [route lifetime](#) (p.51) option is used.

### 6.10.4 Example

```
table radv_routes;                                # Manually configured routes go here

protocol static {
    table radv_routes;

    route 2001:0DB8:4000::/48 unreachable;
    route 2001:0DB8:4010::/48 unreachable;

    route 2001:0DB8:4020::/48 unreachable {
        ra_preference = RA_PREF_HIGH;
        ra_lifetime = 3600;
    };
}

protocol radv {
    propagate routes yes;                          # Propagate the routes from the radv_routes table
    table radv_routes;
    export all;

    interface "eth2" {
        max ra interval 5;                          # Fast failover with more routers
        managed yes;                                # Using DHCPv6 on eth2
        prefix ::/0 {
            autonomous off; # So do not autoconfigure any IP
        };
    };

    interface "eth*";                               # No need for any other options

    prefix 2001:0DB8:1234::/48 {
        preferred lifetime 0; # Deprecated address range
    };

    prefix 2001:0DB8:2000::/48 {
        autonomous off; # Do not autoconfigure
    };

    rdns 2001:0DB8:1234::10; # Short form of RDNS

    rdns {
        lifetime mult 10;
        ns 2001:0DB8:1234::11;
        ns 2001:0DB8:1234::12;
    };

    dnssl {
        lifetime 3600;
        domain "abc.com";
        domain "xyz.com";
    };
}
```

## 6.11 RIP

### 6.11.1 Introduction

The RIP protocol (also sometimes called Rest In Pieces) is a simple protocol, where each router broadcasts (to all its neighbors) distances to all networks it can reach. When a router hears distance to another network, it increments it and broadcasts it back. Broadcasts are done in regular intervals. Therefore, if some network goes unreachable, routers keep telling each other that its distance is the original distance plus 1 (actually, plus interface metric, which is usually one). After some time, the distance reaches infinity (that's 15 in RIP) and all routers know that network is unreachable. RIP tries to minimize situations where counting to infinity is necessary, because it is slow. Due to infinity being 16, you can't use RIP on networks where maximal distance is higher than 15 hosts.

BIRD supports RIPv1 ([RFC 1058](#)), RIPv2 ([RFC 2453](#)), RIPng ([RFC 2080](#)), and RIP cryptographic authentication ([RFC 4822](#)).

RIP is a very simple protocol, and it has a lot of shortcomings. Slow convergence, big network load and inability to handle larger networks makes it pretty much obsolete. It is still usable on very small networks.

### 6.11.2 Configuration

RIP configuration consists mainly of common protocol options and interface definitions, most RIP options are interface specific.

```

protocol rip [<name>] {
    infinity <number>;
    ecmp <switch> [limit <number>];
    interface <interface pattern> {
        metric <number>;
        mode multicast|broadcast;
        passive <switch>;
        address <ip>;
        port <number>;
        version 1|2;
        split horizon <switch>;
        poison reverse <switch>;
        check zero <switch>;
        update time <number>;
        timeout time <number>;
        garbage time <number>;
        ecmp weight <number>;
        ttl security <switch>; | tx only;
        tx class|dscp <number>;
        tx priority <number>;
        rx buffer <number>;
        tx length <number>;
        check link <switch>;
        authentication none|plaintext|cryptographic;
        password "<text>";
        password "<text>" {
            id <num>;
            generate from "<date>";
            generate to "<date>";
            accept from "<date>";
            accept to "<date>";
            from "<date>";
            to "<date>";
            algorithm ( keyed md5 | keyed sha1 | hmac sha1 | hmac sha256 | hmac
};

```

```
    };  
}
```

**infinity** *number*

Selects the distance of infinity. Bigger values will make protocol convergence even slower. The default value is 16.

**ecmp** *switch* [*limit number*]

This option specifies whether RIP is allowed to generate ECMP (equal-cost multipath) routes. Such routes are used when there are several directions to the destination, each with the same (computed) cost. This option also allows to specify a limit on maximum number of nexthops in one route. By default, ECMP is disabled. If enabled, default value of the limit is 16.

**interface** *pattern* [, ...] { *options* }

Interface definitions specify a set of interfaces on which the protocol is activated and contain interface specific options. See [interface](#) (p. 11) common options for detailed description.

Interface specific options:

**metric** *num*

This option specifies the metric of the interface. When a route is received from the interface, its metric is increased by this value before further processing. Valid values are 1-255, but values higher than infinity has no further meaning. Default: 1.

**mode** *multicast|broadcast*

This option selects the mode for RIP to use on the interface. The default is multicast mode for RIPv2 and broadcast mode for RIPv1. RIPng always uses the multicast mode.

**passive** *switch*

Passive interfaces receive routing updates but do not transmit any messages. Default: no.

**address** *ip*

This option specifies a destination address used for multicast or broadcast messages, the default is the official RIP (224.0.0.9) or RIPng (ff02::9) multicast address, or an appropriate broadcast address in the broadcast mode.

**port** *number*

This option selects an UDP port to operate on, the default is the official RIP (520) or RIPng (521) port.

**version** *1|2*

This option selects the version of RIP used on the interface. For RIPv1, automatic subnet aggregation is not implemented, only classful network routes and host routes are propagated. Note that BIRD allows RIPv1 to be configured with features that are defined for RIPv2 only, like authentication or using multicast sockets. The default is RIPv2 for IPv4 RIP, the option is not supported for RIPng, as no further versions are defined.

**version only** *switch*

Regardless of RIP version configured for the interface, BIRD accepts incoming packets of any RIP version. This option restrict accepted packets to the configured version. Default: no.

**split horizon** *switch*

Split horizon is a scheme for preventing routing loops. When split horizon is active, routes are not regularly propagated back to the interface from which they were received. They are either not propagated back at all (plain split horizon) or propagated back with an infinity metric (split horizon with poisoned reverse). Therefore, other routers on the interface will not consider the router as a part of an independent path to the destination of the route. Default: yes.

**poison reverse** *switch*

When split horizon is active, this option specifies whether the poisoned reverse variant (propagating routes back with an infinity metric) is used. The poisoned reverse has some advantages in faster convergence, but uses more network traffic. Default: yes.

**check zero** *switch*

Received RIPv1 packets with non-zero values in reserved fields should be discarded. This option specifies whether the check is performed or such packets are just processed as usual. Default: yes.

**update time** *number*

Specifies the number of seconds between periodic updates. A lower number will mean faster convergence but bigger network load. Default: 30.

**timeout time** *number*

Specifies the time interval (in seconds) between the last received route announcement and the route expiration. After that, the network is considered unreachable, but still is propagated with infinity distance. Default: 180.

**garbage time** *number*

Specifies the time interval (in seconds) between the route expiration and the removal of the unreachable network entry. The garbage interval, when a route with infinity metric is propagated, is used for both internal (after expiration) and external (after withdrawal) routes. Default: 120.

**ecmp weight** *number*

When ECMP (multipath) routes are allowed, this value specifies a relative weight used for nexthops going through the iface. Valid values are 1-256. Default value is 1.

**authentication** *none|plaintext|cryptographic*

Selects authentication method to be used. **none** means that packets are not authenticated at all, **plaintext** means that a plaintext password is embedded into each packet, and **cryptographic** means that packets are authenticated using some cryptographic hash function selected by option **algorithm** for each key. The default cryptographic algorithm for RIP keys is Keyed-MD5. If you set authentication to not-none, it is a good idea to add **password** section. Default: none.

**password** *"text"*

Specifies a password used for authentication. See [password](#) (p. 12) common option for detailed description.

**ttl security** [*switch* | **tx only**]

TTL security is a feature that protects routing protocols from remote spoofed packets by using TTL 255 instead of TTL 1 for protocol packets destined to neighbors. Because TTL is decremented when packets are forwarded, it is non-trivial to spoof packets with TTL 255 from remote locations.

If this option is enabled, the router will send RIP packets with TTL 255 and drop received packets with TTL less than 255. If this option is set to **tx only**, TTL 255 is used for sent packets, but is not checked for received packets. Such setting does not offer protection, but offers compatibility with neighbors regardless of whether they use ttl security.

For RIPng, TTL security is a standard behavior (required by [RFC 2080](#)) and therefore default value is yes. For IPv4 RIP, default value is no.

**tx class|dscp|priority** *number*

These options specify the ToS/DiffServ/Traffic class/Priority of the outgoing RIP packets. See [tx class](#) (p. 11) common option for detailed description.

**rx buffer** *number*

This option specifies the size of buffers used for packet processing. The buffer size should be bigger than maximal size of received packets. The default value is 532 for IPv4 RIP and interface MTU value for RIPng.

**tx length** *number*

This option specifies the maximum length of generated RIP packets. To avoid IP fragmentation, it should not exceed the interface MTU value. The default value is 532 for IPv4 RIP and interface MTU value for RIPng.

**check link** *switch*

If set, the hardware link state (as reported by OS) is taken into consideration. When the link disappears (e.g. an ethernet cable is unplugged), neighbors are immediately considered unreachable and all routes

received from them are withdrawn. It is possible that some hardware drivers or platforms do not implement this feature. Default: no.

### 6.11.3 Attributes

RIP defines two route attributes:

`int rip_metric`

RIP metric of the route (ranging from 0 to infinity). When routes from different RIP instances are available and all of them have the same preference, BIRD prefers the route with lowest `rip_metric`. When a non-RIP route is exported to RIP, the default metric is 1.

`int rip_tag`

RIP route tag: a 16-bit number which can be used to carry additional information with the route (for example, an originating AS number in case of external routes). When a non-RIP route is exported to RIP, the default tag is 0.

### 6.11.4 Example

```
protocol rip {
    import all;
    export all;
    interface "eth*" {
        metric 2;
        port 1520;
        mode multicast;
        update time 12;
        timeout time 60;
        authentication cryptographic;
        password "secret" { algorithm hmac sha256; };
    };
}
```

## 6.12 Static

The Static protocol doesn't communicate with other routers in the network, but instead it allows you to define routes manually. This is often used for specifying how to forward packets to parts of the network which don't use dynamic routing at all and also for defining sink routes (i.e., those telling to return packets as undeliverable if they are in your IP block, you don't have any specific destination for them and you don't want to send them out through the default route to prevent routing loops).

There are five types of static routes: 'classical' routes telling to forward packets to a neighboring router, multipath routes specifying several (possibly weighted) neighboring routers, device routes specifying forwarding to hosts on a directly connected network, recursive routes computing their nexthops by doing route table lookups for a given IP, and special routes (sink, blackhole etc.) which specify a special action to be done instead of forwarding the packet.

When the particular destination is not available (the interface is down or the next hop of the route is not a neighbor at the moment), Static just uninstalls the route from the table it is connected to and adds it again as soon as the destination becomes adjacent again.

There are three classes of definitions in Static protocol configuration – global options, static route definitions, and per-route options. Usually, the definition of the protocol contains mainly a list of static routes.

Global options:

`check link switch`

If set, hardware link states of network interfaces are taken into consideration. When link disappears (e.g. ethernet cable is unplugged), static routes directing to that interface are removed. It is possible that some hardware drivers or platforms do not implement this feature. Default: off.

`igp table name`

Specifies a table that is used for route table lookups of recursive routes. Default: the same table as the protocol is connected to.

Route definitions (each may also contain a block of per-route options):

`route prefix via ip`

Static route through a neighboring router. For link-local next hops, interface can be specified as a part of the address (e.g., `via fe80::1234%eth0`).

`route prefix multipath via ip [weight num] [bfd switch] [via ...]`

Static multipath route. Contains several nexthops (gateways), possibly with their weights.

`route prefix via "interface"`

Static device route through an interface to hosts on a directly connected network.

`route prefix recursive ip`

Static recursive route, its nexthop depends on a route table lookup for given IP address.

`route prefix blackhole|unreachable|prohibit`

Special routes specifying to silently drop the packet, return it as unreachable or return it as administratively prohibited. First two targets are also known as `drop` and `reject`.

Per-route options:

`bfd switch`

The Static protocol could use BFD protocol for next hop liveness detection. If enabled, a BFD session to the route next hop is created and the static route is BFD-controlled – the static route is announced only if the next hop liveness is confirmed by BFD. If the BFD session fails, the static route is removed. Note that this is a bit different compared to other protocols, which may use BFD as an advisory mechanism for fast failure detection but ignores it if a BFD session is not even established.

This option can be used for static routes with a direct next hop, or also for for individual next hops in a static multipath route (see above). Note that BFD protocol also has to be configured, see [BFD](#) (p.24) section for details. Default value is no.

`filter expression`

This is a special option that allows filter expressions to be configured on per-route basis. Can be used multiple times. These expressions are evaluated when the route is originated, similarly to the import filter of the static protocol. This is especially useful for configuring route attributes, e.g., `ospf_metric1 = 100`; for a route that will be exported to the OSPF protocol.

Static routes have no specific attributes.

Example static config might look like this:

```
protocol static {
    table testable;           # Connect to a non-default routing table
    check link;              # Advertise routes only if link is up
    route 0.0.0.0/0 via 198.51.100.130; # Default route
    route 10.0.0.0/8 multipath # Multipath route
        via 198.51.100.10 weight 2
        via 198.51.100.20 bfd # BFD-controlled next hop
        via 192.0.2.1;
    route 203.0.113.0/24 unreachable; # Sink route
    route 10.2.0.0/24 via "arc0"; # Secondary network
    route 192.168.10.0/24 via 198.51.100.100 {
        ospf_metric1 = 20; # Set extended attribute
    }
    route 192.168.10.0/24 via 198.51.100.100 {
        ospf_metric2 = 100; # Set extended attribute
    }
}
```

```
        ospf_tag = 2;          # Set extended attribute
        bfd;                  # BFD-controlled route
    }
}
```

# Chapter 7: Conclusions

## 7.1 Future work

Although BIRD supports all the commonly used routing protocols, there are still some features which would surely deserve to be implemented in future versions of BIRD:

- Opaque LSA's
- Route aggregation and flap dampening
- Multipath routes
- Multicast routing protocols
- Ports to other systems

## 7.2 Getting more help

If you use BIRD, you're welcome to join the bird-users mailing list ([bird-users@network.cz](mailto:bird-users@network.cz)) where you can share your experiences with the other users and consult your problems with the authors. To subscribe to the list, visit [http://bird.network.cz/?m\\_list](http://bird.network.cz/?m_list). The home page of BIRD can be found at <http://bird.network.cz/>.

BIRD is a relatively young system and it probably contains some bugs. You can report any problems to the bird-users list and the authors will be glad to solve them, but before you do so, please make sure you have read the available documentation and that you are running the latest version (available at [bird.network.cz/pub/bird](http://bird.network.cz/pub/bird)). (Of course, a patch which fixes the bug is always welcome as an attachment.)

If you want to understand what is going inside, Internet standards are a good and interesting reading. You can get them from <ftp.rfc-editor.org> (or a nicely sorted version from [atrey.karlin.mff.cuni.cz/pub/rfc](http://atrey.karlin.mff.cuni.cz/pub/rfc)).

*Good luck!*