GNU Dico Manual

version 2.4, 21 November 2016

Sergey Poznyakoff.

Published by the Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Dédié à la mémoire de Jacques Brel.

Short Contents

Table of Contents

Preface

A *dictionary server* is a program that provides dictionary services to other computers using the client-server model. The dictionary services include listing the available databases, searching for a specific term in one or more databases, displaying the definitions found, etc.

GNU Dico is an implementation of dictionary server, which supports a wide variety of database formats and is easily extensible using several scripting languages. Apart from the server, the package contains a console dictionary client and a window-based browser for GCIDE dictionary.

1 Overview

A dictionary server operates on a set of *databases*. Each database contains a set of *headwords* with corresponding *articles*, therefore it can be regarded as a dictionary, in which articles supply definitions (or translations) for headwords.

The server offers facilities for searching headwords in the databases and for fetching articles from them.

This chapter provides an overview of the dictionary protocol and defines basic terms and notions used throughout this manual.

When describing the protocol, the following typographic conventions are used: the data sent by the client are prefixed with '`C:`' and the data sent in response by the server are prefixed with '`S:`'.

Each database has a unique name – a string of characters that serves to identify this particular database in a set of available databases. Two more pieces of textual data are associated with a database. The *database information* string (or *info*, for short), supplies a short description of the database. It is a sentence, tersely describing the database, e.g. '`English-German Dictionary`'. The *database description* provides a full description of the dictionary, with author credits and copyright information. The length of this description is not limited.

Both pieces of information can be requested by the remote user. The command `SHOW DB` lists all available databases along with their descriptions:

```
C: SHOW DB
S: 110 3 databases present
S: jargon "Jargon File (4.3.1, 29 Jun 2001)"
S: deu-eng "German-English Freedict dictionary"
S: en-pl-naut "English-Polish dictionary of nautical terms"
S: .
S: 250 ok
```

Each line of output lists a name of the dictionary, and the corresponding description.

The `SHOW INFO` command displays full information about a database, whose name is given as its argument:

```
C: SHOW INFO en-pl-naut
S: 112 information for en-pl-naut
S: English-Polish dictionary of nautical terms
S:
S: Permission is granted to copy, distribute and/or modify
S: this document under the terms of the GNU Free Docu-
S: mentation License, Version 1.2 or any later version
S: published by the Free Software Foundation; with no
S: Invariant Sections, no Front-Cover and Back-Cover Texts
S: .
```

```
S: 250 ok
```

A definition for any given headword can be requested using the `DEFINE` command. It takes two arguments, the name of the database and the headword to look for in that database, e.g.:

```
DEFINE en-pl-naut sprit
```

If the headword is found in the database, its definition is displayed, otherwise a diagnostic message is returned, telling that the headword was not found.

A special mechanism is provided for looking up the headword in a database (or databases). The `MATCH` command returns headwords that match a given string (a *search key*) using a particular *strategy*. In other words, a strategy identifies the algorithm for comparing two strings: a headword and the search key. A strategy is identified by its name. For example, the strategy '`exact`' means literal comparison and returns only those headwords that match the key exactly. The strategy '`prefix`' matches word prefixes. These two strategies are always present. Depending on the configuration, the server may offer other strategies as well. See Appendix A [Available Strategies], page 101, for a complete list of strategies implemented in GNU Dico 2.4.

One of the strategies is selected as a *default strategy*. Usually such strategy tolerates possible typing errors and allows the user to find matching headwords even if he does not know exactly how the word in question is spelled. The default strategy is denoted as '`.`' (a dot).

The `MATCH` command takes three arguments: the name of the database to search, the strategy and the search key. For example:

```
S: MATCH wn prefix sail
C: 152 4 matches found: list follows
C: wn "sail"
C: wn "sail through"
C: wn "sailboat"
C: wn "sailcloth"
C: .
C: 250 Ok
```

Two database names are special. The '`*`' means search in all databases and return all matches. The '`!`' means search in all databases until the match is found in one of them and return only matches from that particular database.

These are basic facilities provided by the DICT protocol. For a complete and detailed description of the protocol, see Appendix B [Dictionary Server Protocol], page 103.

2 Introduction to GNU Dico

GNU Dico is an implementation of DICT dictionary server (described in RFC 2229) and a set of accompanying utilities. The GNU Dico server uses two-layer model. The *protocol layer* is responsible for the correct DICT protocol dialog and is provided by the `dicod` server binary. The *database layer* is responsible for searching and retrieving data from dictionary databases. This layer is provided by external *loadable modules*. Thus, Dico does not impose any specific dictionary database format. A single server can handle databases in various formats, provided that appropriate modules are available. Several database modules are shipped with GNU Dico. The following is a short introductions for some of them. See Chapter 5 [Modules], page 41, for a complete list of available modules with detailed descriptions.

dictorg
: This module provides full support for the format designed by the *DICT development group* (`http://dict.org`). This is a *de facto* standard for DICT databases. A number of dictionary databases in this format are provided by the *FreeDict* project (`http://freedict.org`).

wordnet
: Support for '`WordNet`' databases. WordNet is a lexical database for the English language developed in the Princeton University and distributed under a BSD style license.

gcide
: Support for '`GNU Collaborative International Dictionary of English`'. This dictionary derived from Webster's Revised Unabridged Dictionary, supplemented with some of the definitions from WordNet. It was edited by Patrick J. Cassidy, proofread and supplemented by volunteers from around the world. It is available from `http://gcide.gnu.org.ua`.

guile
: This module provides an interface to Guile, the *GNU's Ubiquitous Intelligent Language for Extensions* (`http://www.gnu.org/software/guile`) and allows you to write Dico modules in Scheme programming language.

python
: This module provides an interface to Python (`http://www.python.org`) and can be used to write Dico modules in it.

outline
: This module handles simple databases in GNU Emacs *outline* format. It is designed mostly for test purposes.

This manual describes how to configure and use the Dico dictionary system. It also describes the API for writing Dico modules in C, Scheme or Python.

3 Building the Package

Building Dico is quite straightforward. You run `./configure`, then `make`, followed by `make install`, and you are done.

Actions the `configure` script performs are controlled by a set of command line options and variables. Some of these options are generic, i.e. common for all packages using the GNU `autoconf` system. For a detailed description of these option see the `INSTALL` file shipped with the sources. Yet another options are specific for Dico. We will describe them in this chapter.

3.1 Default Preprocessor

The runtime configuration system uses `m4` to preprocess the configuration file (see Section 4.3.16 [Preprocessor], page 35), which makes the configuration extremely flexible. We recommend to use GNU m4 as a preprocessor[1]. However, any other implementation of `m4` can be used as well. The `configure` script tries to determine full file name of the preprocessor binary and the necessary command line options. In case it makes a wrong guess, you can instruct it to use a particular preprocessor by using `DEFAULT_PREPROCESSOR` configuration variable. For example, the following `configure` invocation instructs it to use `/usr/local/bin/gm4`:

```
$ ./configure DEFAULT_PREPROCESSOR="/usr/local/bin/gm4 -s"
```

Note the use of the `-s` preprocessor option. It instructs `m4` to produce line directives which help `dicod` produce correct diagnostics about eventual configuration errors. Unless your `m4` implementation does not have this feature, we recommend to always use it in `DEFAULT_PREPROCESSOR` value.

Finally, if you do not wish to use preprocessor at all, you can disable it using `--without-preprocessor` option to `configure`.

3.2 Default Server

Unless given an explicit dictionary server, the `dico` client program attempts to connect to the server '`dict://dico.gnu.org.ua`'. You may change this default by defining the `DEFAULT_DICT_SERVER` variable. For example, the following command line selects '`dict.org`' as the default server:

$ ./configure DEFAULT_DICT_SERVER=dict.org

The value of the `DEFAULT_DICT_SERVER` variable can be either a hostname or IP address of the server. It can also be followed by a colon and a port specification, either as a decimal number or as a service name from `/etc/services`.

[1] http://www.gnu.org/software/m4

3.3 Guile Support

The *GNU's Ubiquitous Intelligent Language for Extensions*, or *Guile*[2] can be used to write database modules for GNU Dico. This requires Guile version 1.8.4 or newer. The `configure` script will probe for the presence of Guile on your system and automatically enable its use if its version number is high enough.

If you do not wish to use Guile, use `--without-guile` to disable it.

3.4 Python Support

The support for Python (`http://www.python.org`) is enabled automatically if `configure` detects that Python version 2.5 or later is installed on your machine.

If you do not wish to use Python, use `--without-python` to disable it.

3.5 Other Configure Settings

The `dicod` daemon uses `syslogd` for diagnostics. The default syslog facility can be set using `LOG_FACILITY` configuration variable. Its allowed arguments are '`user`', '`daemon`', '`auth`', '`authpriv`', '`mail`', '`cron`', and '`local0`' through '`local7`'. Case is not significant. In addition, these words can be prefixed with '`log_`'.

By default, the '`daemon`' facility is used.

[2] `http://www.gnu.org/software/guile`.

4 The dicod daemon.

The main component of GNU Dico is the `dicod` daemon. It is responsible for serving client requests and for coordinating the work of dictionary modules.

There are two *operation modes*: '`daemon`' and '`inetd`'.

4.1 Daemon Operation Mode

The '`daemon`' mode is enabled by `mode daemon` statement in the configuration file (see [mode statement], page 14). It is also the default mode. In daemon mode `dicod` listens for incoming requests on one or several network interfaces. Unless the `--foreground` option is specified, it detaches itself from the controlling terminal and switches to background (becomes a *daemon*). When an incoming connection arrives, it forks a subprocess for handling it.

In this mode the following signals cause `dicod` to terminate: '`SIGTERM`', '`SIGQUIT`', and '`SIGINT`'. The '`SIGHUP`' signal causes the program to restart. This works only if both the program name and its configuration file name (if given using `--config` option) are absolute file names.

Upon receiving '`SIGHUP`', `dicod` first verifies if the configuration file does not contain fatal errors. To do that, the program executes a copy of itself with the `--lint` option (see [–lint], page 37) and analyzes its return code. Only if this check passes, `dicod` restarts itself. This ensures that the daemon will not terminate due to unnoticed errors in its configuration file.

Upon receiving '`SIGTERM`', '`SIGQUIT`', or '`SIGINT`', the program stops accepting incoming requests and sends the '`SIGTERM`' signal to all active subprocesses. Then it waits a predefined amount of time for all processes to terminate (see [shutdown-timeout], page 16). Any subprocesses that do not terminate after this time are sent the '`SIGKILL`' signal. Then, the database modules are unloaded and `dicod` terminates.

Several command line options are provided that modify the behavior of `dicod` in this mode. These options are mainly designed for debugging and error-hunting purposes.

The `--foreground` option instructs the server to remain attached to the controlling terminal and stay in the foreground. It is often used with `--stderr` option, which instructs `dicod` to output all diagnostic to the standard error output, instead of syslog which is used by default.

4.2 Inetd Operation Mode

In '`inetd`' operation mode `inetd` receives requests from standard input and sends its replies to the standard output. This mode is enabled by `mode inetd` statement (see [mode statement], page 14) in configuration file, or by the `--inetd` command line option (see [–inetd], page 37). This mode is usually used when invoking `dicod` from `inetd.conf` file, as in example below:

```
dict  stream  tcp  nowait  nobody /usr/bin/dicod --inetd
```

4.3 Configuration

Upon startup, `dicod` reads its settings and database definitions from a *configuration file* `dicod.conf`. By default it is located in *$sysconfidr* (i.e., in most cases `/usr/local/etc`, or `/etc`), but an alternative location may be specified using the `--config` command line option (see [–config], page 38).

If any errors are encountered in the configuration file, the program reports them on the standard error and exits with a non-zero status.

To test the configuration file without starting the server, use the `--lint` (`-t`) command line option. It causes `dicod` to check its configuration file and exit with status 0 if no errors were detected, and with status 1 otherwise.

Before parsing, the configuration file is preprocessed using `m4` (see Section 4.3.16 [Preprocessor], page 35). To examine the preprocessed configuration without actually parsing it, use the `-E` command line option. To avoid preprocessing it, use the `--no-preprocessor` option.

The rest of this section describes configuration file syntax in detail. You can receive a concise summary of all configuration directives any time by running `dicod --config-help`.

4.3.1 Configuration File Syntax

A `dicod` configuration consists of statements and comments.

There are three classes of lexical tokens: keywords, values, and separators. Blanks, tabs, newlines and comments, collectively called *white space* are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent keywords and values.

4.3.1.1 Comments

Comments may appear anywhere where white space may appear in the configuration file. There are two kinds of comments: single-line and multi-line comments. *Single-line* comments start with '`#`' or '`//`' and continue to the end of the line:

```
# This is a comment
// This too is a comment
```

Multi-line or *C-style* comments start with the two characters '`/*`' (slash, star) and continue until the first occurrence of '`*/`' (star, slash).

Multi-line comments cannot be nested.

4.3.1.2 Pragmatic Comments

Pragmatic comments are similar to usual comments, except that they cause some changes in the way the configuration is parsed. Pragmatic comments begin with a '`#`' sign and end with the next physical newline character. As of GNU Dico version 2.4, the following pragmatic comments are understood:

`#include <`*file*`>`
`#include` *file*
: Include the contents of the *file*. If *file* is an absolute file name, both forms are equivalent. Otherwise, the form with angle brackets searches for the file in the *include search path*, while the second one looks for it in the current working directory first, and, if not found there, in the include search path.

  The default include search path is:

  1. *prefix*`/share/dico/2.4/include`
  2. *prefix*`/share/dico/include`

  where *prefix* is the installation prefix.

  New directories can be appended in front of it using `-I` (`--include-dir`) command line option (see [–include-dir], page 39).

`#include_once <`*file*`>`
`#include_once` *file*
: Same as `#include`, except that, if the *file* has already been included, it will not be included again.

`#line` *num*
`#line` *num* `"`*file*`"`
: This line causes `dicod` to believe, for purposes of error diagnostics, that the line number of the next source line is given by *num* and the current input file is named by *file*. If the latter is absent, the remembered file name does not change.

`#` *num* `"`*file*`"`
: This is a special form of `#line` statement, understood for compatibility with the C preprocessor.

In fact, these statements provide a rudimentary preprocessing features. For more sophisticated ways to modify configuration before parsing, see Section 4.3.16 [Preprocessor], page 35.

4.3.1.3 Statements

A *simple statement* consists of a keyword and a value separated by any amount of whitespace. It is terminated with a semicolon (‘`;`’), unless the value is a *here-document* (see below), in which case semicolon is optional.

Examples of simple statements:

```
timing yes;
access-log-file /var/log/access_log;
```

A *keyword* begins with a letter and may contain letters, decimal digits, underscores (‘`_`’) and dashes (‘`-`’). Examples of keywords are: ‘`group`’, ‘`identity-check`’.

A *value* can be one of the following:

number
: A number is a sequence of decimal digits.

boolean
: A boolean value is one of the following: '`yes`', '`true`', '`t`' or '`1`', meaning *true*, and '`no`', '`false`', '`nil`', '`0`' meaning *false*.

unquoted string
: An unquoted string may contain letters, digits, and any of the following characters: '`_`', '`-`', '`.`', '`/`', '`@`', '`*`', '`:`'.

quoted string
: A quoted string is any sequence of characters enclosed in double-quotes ('`"`'). A backslash appearing within a quoted string introduces an *escape sequence*, which is replaced with a single character according to the following rules:

Sequence	Replaced with
\a	Audible bell character (ASCII 7)
\b	Backspace character (ASCII 8)
\f	Form-feed character (ASCII 12)
\n	Newline character (ASCII 10)
\r	Carriage return character (ASCII 13)
\t	Horizontal tabulation character (ASCII 9)
\v	Vertical tabulation character (ASCII 11)
\\	A single backslash ('\')
\"	A double-quote.

Table 4.1: Backslash escapes

In addition, the sequence '`\`*`newline`*' is removed from the string. This allows you to split long strings over several physical lines, e.g.:

```
"a long string may be\
 split over several lines"
```

If the character following a backslash is not one of those specified above, the backslash is ignored and a warning is issued.

Two or more adjacent quoted strings are concatenated, which gives another way to split long strings over several lines to improve readability. For instance, the following fragment produces the same result as the example above:

```
"a long string may be"
" split over several lines"
```

Here-document
: A *here-document* is a special construct that allows the user to introduce strings of text containing embedded newlines.

The `<<word` construct instructs the parser to read all the following lines up to the line containing only *word*, with possible trailing blanks. Any lines thus read are concatenated together into a single string. For example:

```
<<EOT
A multiline
string
EOT
```

The body of a here-document is interpreted the same way as a double-quoted string, unless *word* is preceded by a backslash (e.g. '`<<\EOT`') or enclosed in double-quotes, in which case the text is read as is, without interpretation of escape sequences.

If *word* is prefixed with `-` (a dash), then all leading tab characters are stripped from input lines and the line containing *word*. Furthermore, if `-` is followed by a single space, all leading whitespace is stripped from them. This allows for indenting here-documents in a natural fashion. For example:

```
<<- TEXT
    All leading whitespace will be
    ignored when reading these lines.
TEXT
```

It is important that the terminating delimiter be the only token on its line. The only exception to this rule is allowed if a here-document appears as the last element of a statement. In this case a semicolon can be placed on the same line with its terminating delimiter, as in:

```
help-text <<-EOT
        A sample help text.
EOT;
```

list

A *list* is a comma-separated sequence of values. Lists are delimited by parentheses. The following example shows a statement whose value is a list of strings:

```
capability (mime,auth);
```

In any case where a list is appropriate, a single value is allowed without being a member of a list: it is equivalent to a list whose only member is that value. This means that, e.g. '`capability mime;`' is equivalent to '`capability (mime);`'.

A *block statement* introduces a logical group of another statements. It consists of a keyword, followed by an optional value, and a sequence of statements enclosed in curly braces, as shown in the example below:

```
load-module outline {
        command "outline";
}
```

The closing curly brace may be followed by a semicolon, although this is not required.

4.3.2 Server Settings

Server settings control how `dicod` is executed on the server machine.

`user` *`string`* [Configuration]

Run with the privileges of this user. `Dicod` does not require root privileges, so it is recommended to always use this statement when running `dicod` in daemon mode (see Section 4.1 [Daemon Mode], page 9). The argument is either a user name, or UID prefixed with a plus sign.

Example:

```
user nobody;
```

`group` *`list`* [Configuration]

If `user` is given, `dicod` will drop all supplementary groups and switch to the principal group of that user. Sometimes, however, it may be necessary to retain one or more supplementary groups. For example, this might be necessary to access dictionary databases. The `group` statement retains the supplementary groups listed in *list*. Each group can be specified either by its name or by its GID number, prefixed with '`+`', e.g.:

```
user nobody;
group (man, dict, +88);
```

This statement is ignored if `user` statement is not present or if `dicod` is running in inetd mode. See Section 4.2 [Inetd Mode], page 9.

`mode` *`enum`* [Configuration]

Sets server operation mode. The argument is one of:

daemon — Run in daemon mode. See Section 4.1 [Daemon Mode], page 9, for a detailed description.

inetd — Run in inetd mode. See Section 4.2 [Inetd Mode], page 9, for a detailed description.

This statement is overridden by the `--inetd` command line option. See [–inetd], page 37.

`listen` *`list;`* [Configuration]

Specify the IP addresses and ports to listen on in daemon mode. By default, `dicod` will listen on port 2628 on all existing interfaces. Use the `listen` statement to abridge the list of interfaces to listen on, or to change the port number.

Elements of *list* can have the following forms:

host:*port* — Specifies an IP (version 4 or 6) socket to listen on. The *host* part is either an IPv4 in "dotted-quad" notation, or an IPv6 address in square brackets, or a host name. In the latter case,

dicod will listen on all IP addresses corresponding to its 'A' or 'AAAA' DNS records.

The *port* part is either a numeric port number or a symbolic service name which is found in /etc/services file.

Either of the two parts may be omitted. If *host* is omitted, dicod will listen on all interfaces. If *port* is omitted, it defaults to 2628. In this case the colon may be omitted, too.

Examples:

```
listen dict.example.org:2628;
listen 198.51.100.10;
listen [2001:DB8::11];
listen :2628;
```

inet://*host*:*port*
inet4://*host*:*port*

Listen on IPv4 socket. The *host* is either an IP address or a host name. In the latter case, dicod will start listening on all IP addresses from the 'A' records for this host.

Either *host* or *port* (but not both) can be omitted. Missing *host* defaults to IPv4 addresses on all available network interfaces, and missing *port* defaults to 2628.

Example:

```
listen inet4://198.51.100.10;
```

inet6://*host*:*port*

Listen on IPv6 socket. The *host* is either an IPv6 address in square brackets, or a host name. In the latter case, dicod will start listening on all IP addresses from the 'AAAA' records for this host.

Either *host* or *port* (but not both) can be omitted. Missing *host* defaults to IPv6 addresses on all available network interfaces, and missing *port* defaults to 2628.

Example:

```
listen inet6://[2001:DB8::11];
```

filename
unix://*filename*

Specifies the name of a UNIX socket to listen on. *Filename* must be an absolute file name of the socket.

`pidfile` *`string`* [Configuration]

Store PID of the master process in this file. Default is *`localstatedir`*`/run/dicod.pid`. Notice that the access bits of this default directory may be insufficient for dicod to write there after dropping root privileges (see [user statement], page 14). One solution to

this is to create a subdirectory with the same owner as given by `user` statement and to point the PID file there:

```
pidfile /var/run/dict/dicod.pid;
```

Another solution is to make PID directory group-writable and to add the owner group to the `group` statement (see [group statement], page 14).

`max-children` *number* [Configuration]
Sets maximum number of sub-processes that can run simultaneously. This is equivalent to the number of clients that can simultaneously use the server. The default is 64 sub-processes.

`inactivity-timeout` *number* [Configuration]
Set inactivity timeout to the *number* of seconds. The server disconnects automatically if the remote client has not sent any command within this number of seconds. Setting timeout to 0 disables inactivity timeout (the default).

This statement along with `max-children` allows you to control the server load.

`shutdown-timeout` *number* [Configuration]
When the master server is shutting down, wait this number of seconds for all children to terminate. Default is 5 seconds.

`identity-check` *boolean* [Configuration]
Enable identification check using AUTH protocol (RFC 1413). The received user name or UID can be shown in access log using the `%l` conversion (see Section 4.3.8 [Access Log], page 24).

`ident-keyfile` *string* [Configuration]
Use encryption keys from the named file to decrypt AUTH replies encrypted using DES.

`ident-timeout` *number* [Configuration]
Set timeout for AUTH input/output operation to *number* of seconds. Default timeout is 3 seconds.

4.3.3 Authentication

The server may be configured to request authentication in order to make some databases or some additional information available to the user. Another possible use of authentication is to minimize resource utilization on the server machine.

GNU Dico supports two types of authentication: the traditional APOP-style authentication (see Section B.2.5 [AUTH], page 107) and a more advanced SASL authentication. The latter is described separately, see Section 4.3.4 [SASL], page 20.

Authentication setup is simple: first, you define a user authentication database, then you enable it by declaring `auth` server capability (see Section 4.3.10 [Capabilities], page 28):

```
capability auth;
```

User authentication database keeps, for each user name, the corresponding plain text password, and, optionally, the names of groups this user belongs to. Notice, that due to the specifics of DICT authentication scheme (see Section B.2.5 [AUTH], page 107), user passwords are stored in plain text, therefore special care must be taken to protect the contents of your authentication database from compromise.

The database is defined using the `user-db` block statement:

`user-db` *url* [Configuration]
Declare user authentication database.

Dico's authentication is designed so that various authentication database formats can easily be added. A database is identified by its URL, or *Universal Resource Locator*. It consists of the following parts (square brackets denoting optional ones):

type`://[[`*user*`[:`*password*`]@]`*host*`]/`*path*`[`*params*`]`

type A *database type*, or format. See below for a list of available database formats.

user User name necessary to access the database.

password User password necessary to access the database.

host Domain name or IP address of a machine running the database.

path A *path* to the database. The exact meaning of this element depends on the database protocol. It is described in detail when discussing the particular database protocols.

params A list of protocol-dependent parameters. Each parameter is of the form *`keyword=name`*, multiple parameters are separated with semicolons.

If the underlying mechanism requires some additional configuration data that cannot be supplied in an URL, these are passed to it using the following statement:

`options` *string* [user-db conf]
The argument is treated as an opaque string and passed to the authentication '`open`' procedure verbatim. Its exact meaning depends on the type of the database.

The URL defines how the database is accessed. Another important point is where to get the user data from. This is specified by the following two sub-statements:

`password-resource` *arg* [user-db conf]
A database resource which returns the user's password.

`group-resource` *arg* [user-db conf]
A database resource which returns the list of groups this user is member of.

The exact semantics of the *database resource* depends on the type of database being used. For flat text databases, it means the name of a text file that contains these data, for SQL databases, the resource is an SQL query, etc. Below we will discuss URLs and resources used by each database type.

To summarize, the authentication database is defined as:

```
# Define user database for authentication.
user-db url {
  # Additional configuration options.
  options string;

  # Name of a password resource.
  password-resource resource;

  # Name of the resource returning user group information.
  group-resource resource;
}
```

4.3.3.1 Text Authentication Database

A text authentication database consists of one or two flat text files — a *password file*, which contains user passwords, and a *group file*, which contains user groups. The latter is optional. Both files have the same format:

- Empty lines are ignored.
- Any text from '`#`' to the end of the line is ignored.
- Non-empty lines consist of two fields, separated by any amount of white space. The first field is the user name. It serves as a search key in the database. The second field is the requested resource.

Record keys in a password file must be unique, i.e. no two records may contain the same first field. The group file may contain multiple records with the same key. For example:

```
$ grep smith pass
smith guessme
$ grep smith group
smith user
smith timing
smith tester
```

This means that user '`smith`' has password '`guessme`' and is a member of three groups: '`user`', '`timing`' and '`tester`'.

A URL of a text database begins with '`text`' and contains only the *path* element, which gives the name of the directory where the database files reside.

The name of a password file is given by the password-resource statement. The name of a group file is given by the group-resource statement.

For example, if user passwords are kept in the file passwd, user groups are kept in the file user, and both files reside in /var/db/dico directory, then the appropriate database configuration will be:

```
user-db text:///var/db/dico {
  password-resource passwd;
  group-resource group;
}
```

4.3.3.2 LDAP Databases.

To configure LDAP user database, you need first to load the 'ldap' module (see Section 5.13 [ldap], page 66):

```
load-module ldap;
```

The URL of the database is: 'ldap://*host*[:*port*]', where *host* is the host name or IP address of the LDAP server, and optional *port* specifies the port number it is listening on (by default, port 389 is assumed).

The password-resource statement specifies the name of an attribute containing the password, and the group-resource supplies the name of the attribute with the group name.

Additional configuration data are supplied in the options statement, whose argument is a whitespace-separated list of assignments:

base=*base*
: Sets base DN.

binddn=*dn*
: Sets the DN to bind as.

passwd=*string*
: Sets the password.

tls=*bool*
: When set to 'yes', enables the use of TLS encryption.

debug=*number*
: Sets OpenLDAP debug level.

user-filter=*filter*
: A LDAP filter to select the objects describing this user. Any occurrence of '$user' in *filter* is replaced with the actual user name, as obtained during the authentication. This *variable expansion* occurs much the same way as in shell. In particular, the variable is expanded only unless it is immediately followed by an alphanumeric character. For example, it occurs in:

  ```
  (uid=$user)
  ```

  and

```
(uid=$user.1)
```

But it does not occur in

```
(uid=$users)
```

If it is necessary to expand the variable in such a context, enclose its name in curly braces:

```
(uid=${user}s)
```

`group-filter=`*`filter`*
: A LDAP filter that selects the user groups. The *filter* is expanded as in `user-filter`.

The following example shows a LDAP user database configured for base DN '`example.com`' which uses '`posixAccount`' and '`posixGroup`' objects from '`nis.schema`':

```
user-db "ldap://localhost" {
  password-resource userPassword;
  group-resource cn;
  options "user-filter=(uid=$user) "
          "group-filter=(&(objectClass=posixGroup)"
                       "(memberuid=$user)) "
          "base=dc=example,dc=com";
}
```

A note on password usage is in order here. Most authentication methods require the passwords to be stored in the database in *plain text* form. The use of encrypted passwords (e.g. MD5 or SHA1) is possible only with '`LOGIN`' and '`PLAIN`' GSASL authentication methods.

4.3.4 SASL Authentication

The SASL authentication is available if the server was compiled with GNU SASL.

`sasl` *{ statements }* [Configuration]

This block statement configures SASL authentication. The following is a short summary of its syntax and the available substatements:

```
sasl {
  # Disable SASL mechanisms listed in mech.
  disable-mechanism mech;
  # Enable SASL mechanisms listed in mech.
  enable-mechanism mech;
  # Set service name for GSSAPI and Kerberos.
  service name;
  # Set realm name for GSSAPI and Kerberos.
  realm name;
  # Define groups for anonymous users.
  anon-group group-list;
}
```

The list of available authentication mechanisms is configured using two statements:

`disable-mechanism` *mech* [sasl]
Disables SASL mechanisms listed in *mech*, which is a list of names.

`enable-mechanism` *mech* [sasl]
Enables SASL mechanisms listed in *mech*, which is a list of names.

The server builds a list of available mechanisms using the following algorithm. First, a list of implemented mechanisms is retrieved from the SASL library. If the `enable-mechanism` statement is defined, the resulting list is filtered so that only mechanisms listed in `enable-mechanism` remain. Further, if the `disable-mechanism` statement is defined, the names listed there are removed from the list.

`service` *name* [sasl]
Sets the service name for GSSAPI and Kerberos mechanisms.

`realm` *name* [sasl]
Sets the realm name.

`anon-group` *list* [sasl]
Sets the list of user groups considered anonymous.

The database of user credentials depends on the authentication mechanism used. For GSSAPI or Kerberos it is managed by appropriate servers. Other mechanisms use the standard `dicod` user database configuration (see Section 4.3.3 [Authentication], page 16).

4.3.5 Access Control Lists

Access control lists, or ACLs for short, are lists of permissions that can be applied to certain `dicod` objects. They can be used to control who can connect to the dictionary server and what resources are offered to whom.

An ACL is defined using the `acl` block statement:

```
acl name {
  definitions
}
```

The parameter *name* specifies a unique name for that ACL. This name will be used by another configuration statements to refer to that ACL (See Section 4.3.6 [Security Settings], page 23, and see Section 4.3.12.1 [Database Visibility], page 32).

A part between the curly braces (denoted by *definitions* above), is a list of *access statements*. There are two types of such statements:

`allow` *user-group sub-acl host-list* [ACL]
Allow access to resource.

`deny` *`user-group sub-acl host-list`* [ACL]
Deny access to resource.

All parts of an access statement are optional, but at least one of them must be present.

The *user-group* part specifies which users match this entry. Allowed values are the following:

`all` All users.

`authenticated`
Only authenticated users.

`group` *`group-list`*
Authenticated users which are members of at least one of the groups listed in *group-list*.

The *sub-acl* part, if present, branches to another ACL. The syntax of this group is:

```
acl name
```

where *name* is the name of a previously defined ACL.

Finally, the *host-list* group matches client IP addresses. It consists of a `from` keyword followed by a list of *address specifiers*. Allowed address specifiers are:

`any` Matches any client address.

addr Matches if the client IP equals *addr*. The latter may be given either as an IP address or as a host name, in which case it will be resolved and the first of its IP addresses will be used.

addr/*netlen*
Matches if first *netlen* bits from the client IP address equal to *addr*. The network mask length, *netlen* must be an integer number in the range from 0 to 32 for IPv4, and in the range 0 – 128 for IPv6. The address part, *addr*, is as described above.

addr/*netmask*
The specifier matches if the result of logical AND between the client IP address and *netmask* equals to *addr*. The network mask must be specified in a IP address (either IPv4 or IPv6) notation.

filename Matches if connection was received from a UNIX socket *filename*, which must be given as an absolute file name.

To summarize, the syntax of an access statement is:

```
allow|deny [all|authenticated|group group-list]
           [acl name] [from addr-list]
```

where square brackets denote optional parts and vertical bar means '`one of`'.

When an ACL is applied to a particular object, its entries are tried in turn until one of them matches, or the end of the list is reached. If a matched

entry is found, its command verb, allow or deny, defines the result of ACL match. If the end of list is reached, the result is 'allow', unless explicitly specified otherwise.

For example, the following statement defines an ACL named 'common', that allows access for any user connected via local UNIX socket /tmp/dicod.sock or coming from a local network '192.168.10.0/24'. Any authenticated users are allowed, provided that they are allowed by another ACL 'my-nets' (which should have been defined before this definition). Users coming from the network '10.10.0.0/24' are allowed if they authenticate themselves and are members of groups 'dicod' or 'users'. Anybody else is denied access:

```
acl common {
    allow all from ("/tmp/dicod.sock", "192.168.10.0/24");
    allow authenticated acl "my-nets";
    allow group ("dicod", "users") from "10.10.0.0/24";
    deny all;
}
```

See Section 4.3.6 [Security Settings], page 23, for information on how to control daemon security settings.

See Section 4.3.12.1 [Database Visibility], page 32, for a detailed description on how to use ACLs to control access to databases.

4.3.6 Security Settings

This subsection describes configuration settings that control access to various resources served by dicod.

connection-acl *acl-name* [Configuration]

Use ACL *acl-name* to control incoming connections. The ACL itself must be defined before this statement. Using *user-group* (see previous subsection) in this ACL makes no sense, because the authentication itself is performed only after the connection have been established.

```
acl incoming-conn {
   allow from 213.130.0.0/19;
   deny any;
}

connection-acl incoming-conn;
```

show-sys-info *acl-name* [Configuration]

This statement controls whether to show system information in reply to SHOW SERVER command (see Section B.2.3 [SHOW], page 105). The information will be shown only if ACL *acl-name* allows it.

The system information shown includes the following data: name of the package and its version, name of the system where it was built and the kernel version thereof, host name, total operational time of the daemon,

number of subprocesses executed so far and average usage frequency. For example:

```
dicod (dico 2.4) on Linux 2.6.32,
dict.example.net up 99+04:42:58, 19647 forks (686.9/hour)
```

4.3.7 Logging and Debugging

The directives described in this subsection provide basic logging capabilities.

`log-tag` *string* [Configuration]
Prefix syslog messages with this string. By default, the program name is used.

`log-facility` *string* [Configuration]
Sets the syslog facility to use. Allowed values are: '`user`', '`daemon`', '`auth`', '`authpriv`', '`mail`', '`cron`', '`local0`' through '`local7`' (case-insensitive), or a facility number.

`log-print-severity` *boolean* [Configuration]
Prefix diagnostics messages with a string identifying their severity.

`transcript` *boolean* [Configuration]
Controls the transcript of user sessions. If *boolean* is '`true`', the transcript will be output to the logging channel. In the transcript, the lines received from client are prefixed with '`C:`', while those sent in reply are marked with '`S:`'. Here is an excerpt from the transcript output:

```
S: 220 example.net dicod (dico 2.4) <mime.xversion>
  <1645.1212874507@example.net>
C: client "Kdict"
S: 250 ok
C: show db
S: 110 16 databases present
S: afr-deu "Afrikaans-German Freedict dictionary"
S: afr-eng "Afrikaans-English FreeDict Dictionary"
[...]
S: .
S: 250 ok
```

(The first line is split in two to fit in the printed page width.) This option produces lots of output and can significantly slow down the server. Use it only if you are debugging `dicod` or some remote client. Never use it in a production environment.

4.3.8 Access Log

GNU Dico provides a feature similar to Apache's `CustomLog`, which keeps a log of `MATCH` and `DEFINE` requests. To enable this feature, specify the name of the log file using the following directive:

`access-log-file` *string* [Configuration]
Sets access log file name.

```
access-log-file /var/log/dico/access.log;
```

The format of log file entries is defined via the `access-log-format` directive:

`access-log-format` *string* [Configuration]
Sets format string for access log file.

Its argument can contain literal characters, which are copied into the log file verbatim, and *format specifiers*, i.e. special sequences which begin with '`%`' and are replaced in the log file as shown in the table below.

`%%` The percent sign.

`%a` Remote IP-address.

`%A` Local IP-address.

`%B` Size of response in bytes.

`%b` Size of response in bytes in CLF format, i.e. a '`-`' rather than a '`0`' when no bytes are sent.

`%C` Remote client (from the `CLIENT` command, see Section B.2.6 [CLIENT], page 107).

`%D` The time taken to serve the request, in microseconds.

`%d` Request command verb in abbreviated form, suitable for use in URLs, i.e. '`d`' for `DEFINE`, and '`m`' for `MATCH`. See Section 7.1.2 [urls], page 80.

`%h` Remote host.

`%H` Request command verb (`DEFINE` or `MATCH`).

`%l` Remote logname (from identd, if supplied). This will return a dash unless `identity-check` is set to true. See [identity-check], page 16.

`%m` The search strategy.

`%p` The canonical port of the server serving the request.

`%P` The PID of the child that served the request.

`%q` The database from the request.

`%r` Full request.

`%{`*n*`}R` The *n*th token from the request (*n* is 0-based).

`%s` Reply status. For multiple replies, the form '`%s`' returns the status of the first reply, while '`%>s`' returns that of the last reply.

`%t` Time the request was received in the standard Apache format, e.g.:

```
[04/Jun/2008:11:05:22 +0300]
```

`%{`*format*`}t`
The time, in the form given by *format*, which should be a valid `strftime` format. See Appendix C [Time and Date Formats], page 111, for a detailed description.

The standard '`%t`' format is equivalent to

```
[%d/%b/%Y:%H:%M:%S %z]
```

`%T` The time taken to serve the request, in seconds.

`%u` Remote user from `AUTH` command.

`%v` The host name of the server serving the request. See [hostname directive], page 27.

`%V` Actual host name of the server (in case it was overridden in configuration).

`%W` The word from the request.

For the reference, here is the list of format specifiers that have different meaning than in Apache: '`%C`', '`%H`', '`%m`', '`%q`'. The following format specifiers are unique to `dicod`: '`%d`', '`%{`*n*`}R`', '`%V`', '`%W`'.

The absence of `access-log-format` directive is equivalent to the following statement:

```
access-log-format "%h %l %u %t \"%r\" %>s %b";
```

It was chosen so as to be compatible with Apache access logs and be easily parsable by existing log analyzing tools, such as `webalizer`.

Extending this format string with the client name produces a log format similar to Apache '`combined log`':

```
access-log-format "%h %l %u %t \"%r\" %>s %b \"\" \"%C\"";
```

4.3.9 General Settings

Settings described in this subsection configure the basic behavior of the DICT daemon.

`initial-banner-text` *string* [Configuration]
Display the *string* in the textual part of the initial server reply.

When connection is established, the server sends an *initial reply* to the client, that looks like in the example below:

```
220 example.org <auth.mime> <520.1212912026@example.org>
```

See Section B.1 [Initial Reply], page 103, for a detailed description of its parts.

The part of this reply after the host name is modifiable and can contain arbitrary text. You can use `initial-banner-text` to append any additional information there. Note, that *string* may not contain newlines or angle brackets. For example:

```
initial-banner-text "Please authenticate yourself,";
```

This statement produces the following initial reply (split over two lines for readability):

```
220 example.org Please authenticate yourself,
  <auth.mime> <520.1212912026@Texample.org>
```

`hostname` *string* [Configuration]

Sets the hostname. By default, the server determines it automatically. If, however, it makes a wrong guess, you can fix it using this directive.

The server hostname is used, among others, in the initial reply after '220' code (see above) and may also be displayed in the access log file using the '%v' escape (see Section 4.3.8 [Access Log], page 24).

`server-info` *string* [Configuration]

Sets the server description to be shown in reply to `SHOW SERVER` (see Section B.2.3 [SHOW], page 105) command.

The first line of the reply, after the usual '114' response line, shows the name of host where the server is running. If the settings of `show-sys-info` (see Section 4.3.6 [Security Settings], page 23) permit, some additional information about the system is printed.

The lines that follow are taken from the `server-info` directive. It is common to specify *string* using "here-document" syntax (see [here-document], page 12), e.g.:

```
server-info <<EOT
Welcome to the FOO dictionary service.

Contact <dict@foo.example.org> if you have questions or
suggestions.
EOT;
```

`help-text` *string* [Configuration]

Sets the text to be displayed in reply to the HELP command.

The default reply to HELP command displays a list of commands understood by the server with a short description of each.

If the *string* begins with a plus sign, it will be appended to the default reply:

```
help-text <<-EOT
  +
  The commands beginning with an X are extensions.
EOT;
```

If the *string* begins with any other character, except '+', it will replace the default help output. For example:

```
help-text <<-EOT
  There is no help.
  See RFC 2229 for detailed information.
EOT;
```

`default-strategy` *`string`* [Configuration]
Sets the name of the default matching strategy (see Section B.2.2 [MATCH], page 104). By default, Levenshtein matching is used, which is equivalent to

```
default-strategy lev;
```

4.3.10 Server Capabilities

Capabilities are certain server features that can be enabled or disabled at the system administrator's will.

`capability` *`list`* [Configuration]
Requests additional capabilities from the *list*.

The argument to `capability` directive must contain names of existing `dicod` capabilities. These are listed in the following table:

auth — The `AUTH` command is supported. See Section 4.3.3 [Authentication], page 16.

mime — The `OPTION MIME` command is supported. Notice that RFC 2229 requires all servers to support that command, so you should always specify this capability.

xversion — The `XVERSION` command is supported. It is a GNU extension that displays the `dicod` implementation and version number. See Section B.3 [Extended Commands], page 108.

xlev — The `XLEV` command is supported. This command allows the remote party to set and query maximal Levenshtein distance for `lev` matching strategy. See Section B.2.2 [MATCH], page 104. See Section B.3 [Extended Commands], page 108.

The capabilities set using this directive are displayed in the initial server reply (see [initial reply], page 26), and their descriptions are added to the `HELP` command output (unless specified otherwise by the `help-text` statement).

4.3.11 Database Modules and Handlers

A *database module* is an external piece of software designed to handle a particular format of dictionary databases. This piece of software is built as a shared library that `dicod` loads at run time.

A *handler* is an instance of the database module loaded by dicod and configured for a specific database or a set of databases.

Database handlers are defined using the following block statement:

`load-module` *string* `{ ... }` [Configuration]
Create an instance of a database module. The argument specifies a unique name which will be used by subsequent parts of the configuration to refer to this handler. The ellipsis in the description above represents sub-statements. As of Dico version 2.4 only one sub-statement is defined:

`command` *string* [load-module config]
Sets the command line for this handler. It is similar to the shell's command line in that it consists of a name of database module, optionally followed by a whitespace-separated list of its arguments. The name of the module specifies the disk file to load (see below for a detailed description of the loading sequence). Both command name and arguments are passed to the module *initialization function* (see [dico_init, page 69]).

For example:

```
load-module dict {
  command "dictorg dbdir=/var/dicodb";
}
```

This statement defines a handler named '`dict`', which loads the module `dictorg` and passes its initialization function a single argument, '`dbdir=/var/dicodb`'. If the module name is not an absolute file name, as in this example, the loadable module will be searched in the module load path.

A common case is when the module does not require initialization arguments and its command string is the same as its name, e.g.:

```
load-module outline {
  command "outline";
}
```

The configuration syntax provides a shortcut for such usage:

```
load-module outline;
```

If `load-module` is used this way, it accepts a single string or a list of strings as its argument. In the latter case, it loads all modules listed in the argument. For example:

```
load-module (stratall,substr,word);
```

A *module load path* is an internal list of directories which dicod scans in order to find a loadable file name specified in the `command` statement. By default the search order is as follows:

1. Optional *prefix* search directories specified by the `prepend-load-path` directive (see below) and the `--load-dir` (`-L`) command line option.
2. GNU Dico module directory: `$prefix/lib/dico`.

3. Additional search directories specified by the `module-load-path` directive (see below).
4. The value of the environment variable `LTDL_LIBRARY_PATH`.
5. The system dependent library search path (e.g. on GNU/Linux it is defined by the file `/etc/ld.so.conf` and the environment variable `LD_LIBRARY_PATH`).

The value of `LTDL_LIBRARY_PATH` and `LD_LIBRARY_PATH` must be a colon-separated list of absolute directory names, for example '`/usr/lib/mypkg:/lib/foo`'.

In any of these directories, `dicod` first attempts to find and load the given filename. If this fails, it tries to append the following suffixes to it:

1. the libtool archive suffix '`.la`'
2. the suffix used for native dynamic libraries on the host platform, e.g., '`.so`', '`.sl`', etc.

`module-load-path` *list* [Configuration]
This directive adds the directories listed in its argument to the module load path. Example:

```
module-load-path (/usr/lib/dico,/usr/local/dico/lib);
```

`prepend-load-path` *list* [Configuration]
Same as `module-load-path`, but adds directories to the beginning of the module load path.

4.3.12 Databases

Dictionary databases are defined using the `database` block statement.

`database` *{ statements }* [Configuration]
Defines a dictionary database. At least two sub-statements must be defined for each database: `name` and `handler`.

`name` *string* [Database]
Sets the name of this database (a single word). This name will be used to identify this database in DICT commands.

`handler` *string* [Database]
Specifies the handler name for this database and any arguments for it. This handler must be previously defined using the `load-module` statement (see Section 4.3.11 [Handlers], page 28).

For example, the following fragment defines a database named '`en-de`', which is handled by '`dictord`' handler. The handler is passed one argument, `database=en-de`:

```
database {
        name "en-de";
        handler "dictorg database=en-de";
```

```
}
```

More directives are available to fine-tune the database.

`description` *string* [Database]
Supplies a short description, to be shown in reply to `SHOW DB` command. The *string* may not contain new-lines.

Use this statement if the database itself does not supply a description, or if its description is malformed.

In any case, if the `description` directive is specified, its value takes precedence over the description string retrieved from the database itself.

See Section B.2.3 [SHOW], page 105, for a description of `SHOW DB` command.

`info` *string* [Database]
Supplies a full description of the database. This description is shown in reply to `SHOW INFO` (see Section B.2.3 [SHOW], page 105) command. The *string* is usually a multi-line text, so it is common to use here-document syntax (see [here-document], page 12), e.g.:

```
info <<- EOT
   This is a foo-bar dictionary.
   Copyright (C) 2008 foo-bar dict group.
   Distributed under the terms of GNU Free
   Documentation license.
EOT;
```

Use this statement if the database itself does not supply a full description, or if its full description is malformed.

As with `description`, the value of `info` takes precedence over info strings retrieved from the database.

The following two directives control the content type and transfer encoding used when formatting replies from this database if `OPTION MIME` (see Section B.2.4 [OPTION], page 106) is in effect:

`content-type` *string* [Database]
Sets the content type of the reply. E.g.:

```
directory {
   name "foo";
   handler "dictorg";
   content-type "text/html";
   ...
}
```

`content-transfer-encoding` *enum* [Database]
Sets transfer encoding to use when sending MIME replies for this database. Allowed values for *enum* are:

base64 Use BASE64 encoding.

quoted-printable
: Use quoted-printable encoding.

4.3.12.1 Database Visibility

A property called *database visibility* is associated with each dictionary database. It determines whether the database appears in the output of `SHOW DB` command, and takes part in dictionary searches.

By default, all databases are defined as publicly visible. You can, however, limit their visibility on global as well as on per-directory basis. This can be achieved using *visibility ACLs*.

In general, the visibility of a database is controlled by two access control lists: a global visibility ACL and a database visibility ACL. The latter takes precedence over the former.

Both ACLs are defined using the `visibility-acl` statement:

`visibility-acl` *`acl-name`* [Configuration]
: Sets name of the ACL that controls the database visibility. When used in global scope, this statement sets the global visibility ACL. If used within a `database` block, it sets the visibility ACL for that particular database.

Consider the following example:

```
acl glob-vis {
  allow authenticated;
  deny all;
}

acl local-nets {
  allow from (192.168.10.0/24, /tmp/dicod.sock);
}

visibility-acl glob-vis;

database {
  name "terms";
  visibility-acl local-nets;
}
```

In this configuration, the '`terms`' database is visible to everybody coming from the '`192.168.10.0/24`' network and from the UNIX socket `/tmp/dicod.sock`, without authorization. It is not visible to users coming from elsewhere, unless they authenticate themselves.

4.3.13 Strategies and Default Searches

A *default search* is a `MATCH` request with '`*`' or '`!`' as the database argument (see Section B.2.2 [MATCH], page 104). The former means search in all

available databases, the latter means search in all databases until a match is found.

Default searches may be quite expensive and may cause considerable strain on the server. For example, the command `MATCH * priefix ""` returns all entries from all available databases, which would consume a lot of resources both on the server and on the client side.

To minimize harmful effects from such potentially dangerous requests, it is possible to limit the use of certain strategies in default searches.

`strategy` *name* *{ statements }* [Configuration]
Restricts the use of the strategy *name* in default searches.

The *statements* define conditions the 4th argument of a `MATCH` command must match in order to deny the request. The following statements are defined:

`deny-all` *bool* [Configuration]
Unconditionally deny the use of this strategy in default searches.

`deny-word` *list* [Configuration]
Deny this strategy if the search word matches one of the words from *list*.

`deny-length-lt` *number* [Configuration]
Deny if length of the search word is less than *number*.

`deny-length-le` *number* [Configuration]
Deny if length of the search word is less than or equal to *number*.

`deny-length-gt` *number* [Configuration]
Deny if length of the search word is greater than *number*.

`deny-length-ge` *number* [Configuration]
Deny if length of the search word is greater than or equal to *number*.

`deny-length-eq` *number* [Configuration]
Deny if length of the search word is equal to *number*.

`deny-length-ne` *number* [Configuration]
Deny if length of the search word is not equal to *number*.

For example, the following statement denies the use of '`prefix`' strategy in default searches if its argument is an empty string:

```
strategy prefix {
  deny-length-eq 0;
}
```

If the `dicod` daemon is configured this way, it will always return a '`552`' reply on commands `MATCH * prefix ""` or `MATCH ! prefix ""`. However, the use of empty prefix on a concrete database, as in `MATCH eng-deu prefix ""`, will still be allowed.

4.3.14 Tuning

While tuning your server, it is often necessary to get timing information which shows how much time is spent serving certain requests. This can be achieved using the `timing` configuration directive:

`timing` *boolean* [Configuration]

Provide timing information after successful completion of an operation. This information is displayed after the following requests: `MATCH`, `DEFINE`, and `QUIT`. It consists of the following parts:

```
[d/m/c = nd/nm/nc RTr UTu STs]
```

where:

nd Number of processed define requests. It is '`0`' after a `MATCH` request.

nm Number of processed match requests. It is '`0`' after a `DEFINE` request.

nc Number of comparisons made. This value may be inaccurate if the underlying database module is not able to count comparisons.

RT Real time spent serving the request.

UT Time in user space spent serving the request.

ST Time in kernel space spent serving the request.

An example of a server reply with timing information follows:

```
250 Done [d/m/c = 0/63/107265 2.293r 1.120u 0.010s]
```

You can also add timing information to your access log files, see Section 4.3.8 [Access Log], page 24.

4.3.15 Command Aliases

Aliases allow a string to be substituted for a word when it is used as the first word of a command. The daemon maintains a list of aliases that are created using the `alias` configuration file statement:

`alias` *word command* [Configuration]

Creates a new alias.

Aliases are useful to facilitate manual interaction with the server, as they allow the administrator to create abbreviations for some frequently typed commands. For example, the following alias creates new command `d` which is equivalent to `DEFINE *`:

```
alias d DEFINE "*";
```

Aliases may be recursive, i.e. the first word of *command* may refer to another alias. For example:

```
alias d DEFINE;
alias da d "*";
```

This configuration will produce the following expansion:

```
da word ⇒ DEFINE * word
```

To prevent endless loops, recursive expansion is stopped if the first word of the replacement text is identical to an alias expanded earlier.

4.3.16 Using Preprocessor to Improve the Configuration.

Before parsing its configuration file, `dicod` preprocesses it. The built-in preprocessor handles only file inclusion and `#line` statements (see Section 4.3.1.2 [Pragmatic Comments], page 10), while the rest of traditional preprocessing facilities, such as macro expansion, is supported via `m4`, which is used as an external preprocessor.

The detailed description of `m4` facilities lies far beyond the scope of this document. You will find a complete user manual in Section "GNU M4" in *GNU M4 macro processor*. For the rest of this subsection we assume the reader is sufficiently acquainted with `m4` macro processor.

The external preprocessor is invoked with `-s` flag, instructing it to include line synchronization information in its output. This information is then used by the parser to display meaningful diagnostic. An initial set of macro definitions is supplied by the `pp-setup` file, located in *$prefix*`/share/dico/`*version*`/include` directory (where *version* means the version of GNU Dico package).

The default `pp-setup` file renames all `m4` built-in macros so they all start with the prefix '`m4_`'. This is similar to GNU m4 `--prefix-builtin` option, but has an advantage that it works with non-GNU `m4` implementations as well.

As an example of how the use of preprocessor may improve `dicod` configuration, consider the following fragment taken from one of the installations of GNU Dico. This installation offers quite a few Freedict dictionaries. The database definition for each of them is almost the same, except for the dictionary name and eventual description entry for several databases that miss it. To avoid repeating the same text over again, we define the following macro:

```
# defdb(NAME[, DESCR])
# Produce a standard definition for a database NAME.
# If DESCR is given, use it as a description.
m4_define(‘defdb’, ‘
database {
        name "$1";
        handler "dictorg database=$1";m4_dnl
m4_ifelse(‘$2’,,,‘
        description "$2";’)
}
’)
```

It takes two arguments. The first one, NAME, defines the dictionary name visible in the output of `SHOW DB` command. Optional second argument may be used to supply a description string for the databases that miss it.

Given this macro, the database definitions look like:

```
defdb(eng-swa)
defdb(swa-eng)
defdb(afr-eng, Afrikaans-English Dictionary)
defdb(eng-afr, English-Afrikaans Dictionary)
```

4.4 Dicod Exit Codes

Apart from issuing a descriptive error message, `dicod` attempts to indicate the reason of its termination by its error code. As usual, a zero exit code indicates normal termination. The table below summarizes all possible error codes. For each error code, it indicates its decimal value and its symbolic name from `include/sysexits.h` (if available).

0
EX_OK
: Program terminated correctly.

2
: Only child instances of `dicod` exit with this code. It indicates that the child did not receive any ‘`DICT`’ command within the time out interval (see [inactivity-timeout], page 16).

64
EX_USAGE
: The program was invoked incorrectly, e.g. an invalid option was given, or an erroneous argument was supplied to an option.

67
EX_NOUSER
: `Dicod` cannot switch to the privileges of the user it is configured to run as (see [user statement], page 14).

69
EX_UNAVAILABLE
: The server exited due to some error not otherwise described in this table.

70

EX_SOFTWARE
: Some internal software error occurred.

71
EX_OSERR
: Some system error occurred, e.g. the program ran out of memory, or file descriptors, or 'fork' failed, etc.

78
EX_CONFIG
: An error in the configuration file was detected.

4.5 Dicod Invocation

This section summarizes dicod command line options. Options are subdivided in five categories.

4.5.1 Dicod Operation Mode

The following options select the operation mode. Only one of them can be present in the command line:

`-E`
: Preprocess configuration file and exit. See Section 4.3.16 [Preprocessor], page 35.

`-i`
`--inetd`
: Run in inetd mode. See Section 4.2 [Inetd Mode], page 9.

`-r`
`--runtest`
`--test`
: Run unit tests for the module. Arguments following that option are parsed as follows:

    modname [*testargs*] [-- *initargs*]

  where *modname* stands for the name of the module to test, *testargs* are arguments to the dico_run_test function of the module, and *initargs* are module initialization arguments (passed to the dico_init method). Square brackets denote optional parts. Before passing to the corresponding method, both argument lists are augmented by prepending module name as the first element (with index 0).

  This option implies --stderr.

  Use the --load-dir (-L) option (see [–load-dir], page 38), if the module is not located in one of the default load directories (see [load path], page 29).

  See Section 6.4 [Unit Testing], page 76, for a detailed discussion of module unit testing.

`-t`
`--lint` Check configuration file syntax and exit with code '`0`' if it is OK, or with '`78`' if there are errors. See Section 4.3 [Configuration], page 10.

4.5.2 Informational Options

The informational options cause the program to print a selected piece of information and exit. Only one informational option can be used at a time.

`--config-help`
Show a summary of the configuration file syntax and allowed statements. See Section 4.3 [Configuration], page 10.

`-h`
`--help` Display a short command line option summary and exit.

`--usage` List all available command line options and exit.

`--version`
Print program version and exit.

4.5.3 Modifier Options

These options modify the program behavior:

`--config=`*`file`*
Read this configuration file instead of the default *`$sysconfdir`*`/dicod.conf`. See Section 4.3 [Configuration], page 10.

`-f`
`--foreground`
Operate in foreground. See Section 4.1 [Daemon Mode], page 9.

`-L` *`dir`*
`--load-dir=`*`dir`*
Adds *dir* to the beginning of module load path. See [load path], page 29, for detailed discussion.

`-s`
`--single-process`
In daemon mode, process connections in the main process, without starting subprocesses for each connection (see Section 4.1 [Daemon Mode], page 9). This means that the daemon is able to serve only one client at a time. The `--single-process` option is provided for debugging purposes only. Never use it in production environment.

`--stderr` Output the diagnostics to stderr. See Section 4.1 [Daemon Mode], page 9.

`--syslog` After successful startup, output any diagnostic to syslog. This is the default.

4.5.4 Preprocessor Control

The following options control the use of preprocessor. See Section 4.3.16 [Preprocessor], page 35, for a detailed discussion.

`--define=`*`symbol`*`[=`*`value`*`]`
`-D` *`symbol`*`[=`*`value`*`]`
: Define the preprocessor symbol *symbol*. Optional *value* supplies the new symbol value. This option is passed to the preprocessor verbatim.

`-I` *`dir`*
`--include-dir=`*`dir`*
: Add the directory *dir* to the list of directories to be searched for preprocessor include files. See Section 4.3.16 [Preprocessor], page 35.

`--no-preprocessor`
: Do not use external preprocessor. See Section 4.3.16 [Preprocessor], page 35.

`--preprocessor=`*`prog`*
: Use *prog* as a preprocessor for configuration file. The default preprocessor command line is `m4 -s`, unless overridden while configuring the package (see Section 3.1 [Default Preprocessor], page 7).

4.5.5 Debugging Options

`-x`
`--debug=`*`level`*
: Set debug verbosity level. The *level* argument is an integer ranging from '`0`' (no debugging) to '`100`' (maximum debugging information).

`--no-transcript`
: Disable transcript mode. This is the default. Use this option if you wish to temporarily disable transcript mode, enabled in the configuration file (see Section 4.3.7 [Logging and Debugging], page 24).

`-T`
`--transcript`
: Enable session transcript. This instructs `dicod` to log all commands it receives and all responses it sends during the session. Transcript is logged via the default logging channel (see Section 4.3.7 [Logging and Debugging], page 24). If logging via syslog, the '`debug`' priority is used.

  See also Section 7.2.7 [Session Transcript], page 86, for a description of the similar mode in `dico`, the client program.

`--source-info`
: Include source line information in the debugging output.

`--trace-grammar`
: Trace parsing of the config file.

`--trace-lex`
: Trace the configuration file lexer.

5 Modules

GNU Dico comes with a set of loadable modules for handling various database formats and extending the server functionality. Modules are binary loadable files, installed in *$prefix*`/lib/dico`. They are configurable on per-module (see Section 4.3.11 [Handlers], page 28) and per-database (see Section 4.3.12 [Databases], page 30) basis.

In this chapter we will describe the modules included in the distribution of GNU Dico version 2.4.

5.1 `Outline`

The `outline` module supports databases written in *Emacs outline mode*. It is not designed for storing large amounts of data, its purpose rather is to handle small databases that can be composed easily and quickly using the Emacs editor.

The outline mode is described in Section "Outline Mode" in *The Emacs Editor*. In short, it is a usual plain text file, containing *header lines* and *body lines*. Header lines start with one or more stars, the number of starts indicating the nesting level of the heading in the document structure: one star for chapters, two stars for sections, etc. Body lines are anything that is not header lines.

The outline dictionary must have at least a chapter named '`Dictionary`', which contains the dictionary corpus. Within it, each section is treated as a dictionary article, its header line giving the headword, and its body lines supplying the article itself. Apart from this, two more chapters have special meaning. The '`Description`' chapter gives a short description to be displayed on `SHOW DB` command, and the '`Info`' chapter supplies a full database description for `SHOW INFO` output. Both chapters are optional.

All three reserved chapter names are case-insensitive.

To summarize, the structure of an outline database is:

```
* Description
line

* Info
text

* Dictionary

** line
text

[any number of entries follows]
```

As an example of outline format, the GNU Dico package includes Ambrose Bierce's *Devil's Dictionary* in this format, see `examples/devdict.out`.

The initialization of the `outline` module does not require any command line parameters. To declare a database, supply its full file name to the database `handler` directive, as shown in the example below:

```
load-module outline;

database {
   name "devdict";
   handler "outline /var/db/devdict.out";
}
```

5.2 Dictorg

The `dictorg` module supports dictionaries in the format designed by *DICT development group* (`http://dict.org`). Lots of free dictionaries in this format are available from the *FreeDict* project.

A dictionary in this format consists of two files: a *dictionary database file*, named *`name`*`.dict` or *`name`*`.dict.dz` (a compressed form), and an *index file*, which lists article headwords with the corresponding offsets in the database. The index file is named *`name`*`.index`. The common part of these two file names, *name*, is called the *base name* for that dictionary.

An instance of the `dictorg` module is created using the following statement:

```
load-module inst-name {
    command "dictorg [options]";
}
```

where square brackets denote optional part. Valid *options* are the following:

`dbdir=`*`dir`*
: Look for databases in directory *dir*.

`show-dictorg-entries`
: *Dictorg entries* are special database entries that keep some service information, such as database description, etc. Such entries are marked with headwords that begin with '`00-database-`'. By default they are exempt from database look-ups and cannot be retrieved using `MATCH` or `DEFINE` command.

  Using `show-dictorg-entries` removes this limitation.

`sort`
: Sort the database index after loading. This option is designed for use with some databases that have malformed indexes. At the time of this writing the '`eng-swa`' database from *FreeDict* requires this option.

  Using `sort` may considerably slow down initial database loading.

`trim-ws`
: Remove trailing whitespace from dictionary headwords at start up. This might be necessary for some databases.

The values set via these options become defaults for all databases using this module instance, unless overridden in their declarations.

A database that uses this module must be declared as follows:

```
database {
    handler "inst-name database=file [options]";
    ...
}
```

where *inst-name* is the instance name used in the `load-module` declaration above.

The `database` argument specifies the base name of the database. Unless *file* begins with a slash, the value of `dbdir` initialization option is prepended to it. If `dbdir` is not given and *file* does not begin with a slash, an error is signalled.

The *options* above are the same options as described in initialization procedure: `show-dictorg-entries`, `sort`, and `trim-ws`. If used, they override initialization settings for that particular database. Forms prefixed with '`no`' can be used to disable the corresponding option for this database. For example, `notrim-ws` cancels the effect of `trim-ws` used when initializing the module instance.

5.3 Gcide

The `gcide` module provides support for GNU Collaborative International Dictionary of English. This dictionary can be downloaded from `ftp://ftp.gnu.org/gnu/gcide`. It consists of a set of files named from `CIDE.A` through `CIDE.Z`, written using a special markup. See `http://gcide.gnu.org.ua`, for a detailed information about the dictionary.

The `gcide` module is started via the following statement:

```
load-module gcide;
```

The database is initialized as follows:

```
database {
    handler "gcide dbdir=directory [options]";
    ...
}
```

The '`dbdir`' parameter supplies the name of the directory where database files are located. Upon startup, the module scans the dictionary files and creates an index file, named `GCIDE.IDX`, if it does not already exist. The file is created using an ancillary program `idxgcide`, described below. Unless specified otherwise, this file is created in the same directory where the database files are located, therefore the directory must be writable for the user `dicod` is started as.

Other options are:

`idxdir` *directory* [gcide parameter]
Specifies the directory where the `CIDE.IDX` index file resides or should reside.

`index-cache-size` *size* [gcide parameter]
Sets the maximum number of index pages the module keeps in memory simultaneously. The default value is 16. The pages are cached using the *last recently used* algorithm. Raising this value will make dictionary accesses faster at the expense of using more memory.

`index-program` *progname* [gcide parameter]
Specifies the full name of the index program. Usually this option is not needed, because the module is configured to start the `idxgcide` utility from its default location. It is mostly useful for the module developers.

`suppress-pr` [gcide parameter]
This parameter suppresses the output of '`pr`' (pronunciation) tags. According to GCIDE docs, *very few of the pronunciation fields have been filled in*, so it might be reasonable to avoid displaying them at all.

Starting from version 0.51, GCIDE contains the file `INFO`, which provides basic information about the dictionary. The `gcide` module returns contents of this file at the '`SHOW INFO`' request. The first line of this file (with the trailing newline and final point removed) is returned as the short database description.

Here's a full example of a '`gcide`' as used in '`dico.gnu.org.ua`':

```
load-module gcide;

database {
    name "gcide";
    handler "gcide dbdir=/var/dictdb/gcide-0.51 suppress-pr";
    languages-from "en";
    languages-to "en";
}
```

5.3.1 idxgcide

The `idxgcide` utility is used by the `gcide` module to index the GCIDE dictionary. You can start it manually to reindex the database. It can be needed, for example, if you install a modified version of the dictionary. The program is installed in *libexecdir*. The usage is:

```
idxgcide [options] dbdir [idxdir]
```

The only mandatory argument *dbdir* specifies the name of the directory where the GCIDE dictionary is installed. The optional *idxdir* argument specifies the directory for the index file, if it differs from *dbdir*. Available *options* are:

`--debug`
`-d` Debug lexical analyzer.

`--dry-run`
`-n` Do nothing, but print everything. This implies `--verbose`.

`--verbose`
`-v` Increase output verbosity. This option can be specified multiple times, each occurrence increasing the verbosity level by one. By default the utility outputs only errors and warnings. At level one, it prints additionally the names of source files that are being indexed at the moment. At level two (the maximum level implemented at the moment) it outputs each headword being indexed along with its location. This is useful only for debugging.

`--page-size=`*`number`*
`-p` *`number`* Defines the size of index file page. The *number* specifies the size in bytes. The following case-insensitive suffixes can be used: '`k`' ('`kb`'), '`m`' ('`mb`') or '`g`' ('`gb`'), specifying kilobytes, megabytes and gigabytes (ouch!) correspondingly.

The default page size is 10240 bytes.

5.4 Wordnet

WordNet is a lexical database for the English language, created and maintained at the Cognitive Science Laboratory of Princeton University[1]. It groups English words into sets of synonyms called *synsets*, provides short, general definitions, and records the various semantic relations between these synonym sets.

Dico provides a `wordnet` module for reading WordNet lexical database files. The module relies on `libWN`, the support library distributed with the WordNet database.

There is a point worth noticing if you plan to use the WordNet library. Normally, the `libWN` is compiled as a static library with position-dependent code, which makes it difficult (or impossible, on 64-bit architectures) to use from the dynamically-loaded libraries, such as `dicod` modules. So, first of all you will need to rebuild WordNet so that it contains position-independent code. To do so, change to the WordNet source directory and reconfigure it as follows:

```
./configure CFLAGS=-fPIC [other_options]
```

where *other_options* stands for any other options you might wish to pass to configure.

If you are going to run this command in a source directory that has been previously configured, it is advisable to run '`make distclean`' beforehand.

[1] See http://wordnet.princeton.edu/wordnet/, for a detailed information, including links to download.

The `wordnet` module is compiled automatically if the configure script was able to find the library and its header file `wn.h`. If it was not, use the `--with-wordnet` configure option to specify the location where these files can be found. For example, if WordNet was installed using the default procedure, then the following option will do the job:

```
./configure --with-wordnet=/usr/local/WordNet-3.0
```

This command tells Dico to look for WordNet library files in `/usr/local/WordNet-3.0/lib` and for include files in `/usr/local/WordNet-3.0/include`.

A compiled module is loaded using the following statement:

```
load-module wordnet {
    command "wordnet [parameters]";
}
```

Optional parameters are:

`wnhome` *dir* [wordnet module parameter]

Base directory for WordNet files. This is the directory where WordNet was installed. For the `wordnet` module to work, it must contain the `dict` subdirectory with WordNet dictionary files.

If you installed WordNet to `/usr/local/WordNet-3.0`, so that running `ls` on that directory shows you:

```
$ ls /usr/local/WordNet-3.0/
bin/  dict/  doc/  include/  lib/  man/
```

then you would use

```
load-module wordnet {
    command "wordnet wnhome=/usr/local/WordNet-3.0";
}
```

`wnsearchdir` *dir* [wordnet module parameter]

Directory in which the WordNet database has been installed.

Normally, these values are set at compile time and you won't need to override them. The use of these parameters may, however, be necessary if the database was moved or installed in a non-standard location.

One or more WordNet database instances can be defined. They all will be sharing the same database. The reason for having several database instances is that they may have different output options. For example, you may configure one database to return word definitions and another one to act as a thesaurus.

Dico version 2.4 defines the following database parameters:

`pos` *value* [wordnet database parameter]

Select part of speech to be displayed by this database. By default, all parts of speech are displayed. Valid values are:

all Display all parts of speech. This is the default.

noun Display only nouns.

verb Display only verbs.

adj
adjective Display only adjectives.

adv
adverb Display only adverbs.

satellite
adjsat Display only *satellites*.

`merge-defs` [wordnet database parameter]
When specified, this parameter instructs the WordNet database to merge all definitions with the same part of speech into a single definition, which will be returned in the usual dictionary fashion, e.g.:

```
sail
n. 1. a large piece of fabric (usually canvas fabric) by
means of which wind is used to propel a sailing vessel
Synonyms: {canvas}, {canvass}, {sheet}
2. an ocean trip taken for pleasure
Synonyms: {cruise}
3. any structure that resembles a sail
v. 1. traverse or travel on (a body of water); "We sailed
the Atlantic"; "He sailed the Pacific all alone"
2. move with sweeping, effortless, gliding motions
```

By default, each definition is returned as a separate entry.

As an example, the following is the database definition the author uses on his server:

```
database {
    name "WordNet";
    handler "wordnet merge-defs";
    languages-from "en";
    languages-to "en";
    description "WordNet dictionary, version 3.0";
}
```

5.5 Guile

Guile is an acronym for *GNU's Ubiquitous Intelligent Language for Extensions*. It provides a Scheme interpreter conforming to the R5RS language specification and a number of convenience functions. For information about the language, refer to *Revised(5) Report on the Algorithmic Language Scheme*. For a detailed description of Guile and its features, see Section "Overview" in *The Guile Reference Manual*.

The `guile` module provides an interface to Guile that allows for writing GNU Dico modules in Scheme. The module is loaded using the following configuration file statement:

```
load-module mod-name {
   command "guile [options]"
           " init-script=script"
           " init-args=args"
           " init-fun=function";
}
```

The `init-script` parameter specifies the name of a Scheme source file to be loaded in order to initialize the module. The `init-args` parameter supplies additional arguments to the module. They will be accessible to the *`script`* via `command-line` function. This parameter is optional.

The `init-fun` parameter specifies the name of a function that will be invoked to perform initialization of the module and of particular databases. See Section 5.5.2 [Guile Initialization], page 49, for a description of initialization sequence. Optional arguments, *options*, are:

`debug` Enable Guile debugging and stack traces.

`nodebug` Disable Guile debugging and stack traces (default).

`load-path=`*`path`*
: Append directories from *path* to the list of directories which should be searched for Scheme modules and libraries. The *path* must be a list of directory names, separated by colons.

  This option modifies the value of Guile's `%load-path` variable. See Section "Configuration Build and Installation" in *The Guile Reference Manual*.

Guile databases are declared using the following syntax:

```
database {
        name "dbname";
        handler "mod-name [options] cmdline";
}
```

where:

dbname gives the name for this database,

mod-name
: the name given to Guile module in `load-module` statement (see above),

options options that override global settings given in the `load-module` statement. The following options are understood: `init-script`, `init-args`, and `init-fun`. Their meaning is the same as for `load-module` statement (see above), except that they affect only this particular database.

cmdline the command line that will be passed to the Guile `open-db` callback function (see [open-db], page 50).

5.5.1 Virtual Functions

A database handled by the `guile` module is assigned a *virtual function table*. This table is an association list which keeps Scheme *call-back functions* implemented to perform particular tasks on that database. In this list, the `car` of each element contains the name of a function, and its `cdr` gives the corresponding function. The defined function names and their semantics are:

open Open the database.

close Close the database.

descr Return a short description of the database.

info Return a full information about the database.

define Define a word.

match Look up a word in the database.

output Output a search result.

result-count
Return number of entries in the result.

For example, the following is a valid virtual function table:

```
(list (cons "open" open-module)
      (cons "close" close-module)
      (cons "descr" descr)
      (cons "info" info)
      (cons "define" define-word)
      (cons "match" match-word)
      (cons "output" output)
      (cons "result-count" result-count))
```

Apart from a per-database virtual table, there is also a global virtual function table, which supplies entries missing in the former. Both tables are created during the module initialization, as described in the next subsection.

The purposes of particular virtuals functions are described in Section 5.5.3 [Guile API], page 50.

5.5.2 Guile Initialization

The following configuration statement causes loading and initialization of the `guile` module:

```
load-module mod-name {
   command "guile init-script=script"
           " init-fun=function";
}
```

Upon module initialization stage, the module attempts to load the file named *script*. The file is loaded using `primitive-load` call (see Section "Loading" in *The Guile Reference Manual*), i.e. the load paths are not searched, so *script* must be an absolute path name. The `init-fun` parameter supplies the name of the *initialization function*. This Scheme function constructs virtual function tables for the module itself and for each database that uses this module. It must be declared as follows:

```
(define (function arg)
  ...)
```

This function is called several times. First of all, it is called after the *script* is loaded. This time it is given `#f` as its argument, and its return value is saved as a global function table. Then, it is called for each `database` statement that has *mod-name* (used in `load-module` above) in its `handler` keyword, e.g.:

```
database {
   name db-name;
   handler "mod-name ...";
}
```

This time, it is given *db-name* as its argument and the value it returns is stored as the virtual function table for this particular database.

The following example function returns a complete virtual function table:

```
(define-public (my-dico-init arg)
  (list (cons "open" open-module)
        (cons "close" close-module)
        (cons "descr" descr)
        (cons "info" info)
        (cons "lang" lang)
        (cons "define" define-word)
        (cons "match" match-word)
        (cons "output" output)
        (cons "result-count" result-count)))
```

5.5.3 Guile API

This subsection describes callback functions that a Guile database module must provide. Each description begins with the function prototype and its entry in the virtual function table.

Callback functions can be subdivided into two groups: database functions and search functions.

Database callback functions are responsible for opening and closing databases and for returning information about them.

`open-db` *name . args* [Guile Callback]

Virtual table: `(cons "open" open-db)`

Open the database. The argument *name* contains database name as given in the `name` statement of the corresponding `database` block (see

Section 4.3.12 [Databases], page 30). Optional argument *args* is a list of command line parameters obtained from *cmdline* in `handler` statement (see [guile-cmdline], page 48). For example, if the configuration file contained:

```
database {
    name "foo";
    handler "guile db=file 1 no";
}
```

then the `open-db` callback will be called as:

```
(open-db "foo" '("db=file" "1" "no"))
```

The `open-db` callback returns a *database handle*, i.e. an opaque object that will subsequently be used to identify this database. This value, hereinafter named *dbh*, will be passed to another callback functions that need to access the database.

The return value `#f` or `'()` indicates an error.

`close-db` *dbh* [Guile Callback]

Virtual Table: `(cons "close" close-db)`

Close the database. This function is called during the cleanup procedure, before termination of `dicod`. The argument `dbh` is a database handle returned by `open-db`.

The return value from `close-db` is ignored. To communicate errors to the daemon, throw an exception.

`descr` *dbh* [Guile Callback]

Virtual Table: `(cons "descr" descr)`

Return a short textual description of the database, for use in `SHOW DB` output. If there is no description, returns `#f` or `'()`.

The argument *dbh* is a database handle returned by `open-db`.

This callback is optional. If it is not defined, or if it returns `#f` (`'()`), the text from `description` statement is used (see Section 4.3.12 [Databases], page 30). Otherwise, if no `description` statement is present, an empty string will be returned.

`info` *dbh* [Guile Callback]

Virtual Table: `(cons "info" info)`

Return a verbose, eventually multi-line, textual description of the database, for use in `SHOW INFO` output. If there is no description, returns `#f` or `'()`.

The argument *dbh* is a database handle returned by `open-db`.

This callback is optional. If it is not defined, or if it returns `#f` (`'()`), the text from `info` statement is used (see Section 4.3.12 [Databases], page 30).

If there is no `info` statement, the string '`No information available`' is used.

`lang` *dbh* [Guile Callback]
Virtual Table: `(cons "lang" lang)`

Return a `cons` of languages supported by this database: Its `car` is a list of source languages, and its `cdr` is a list of destination languages. For example, the following return value indicates that the database contains translations from English to French and Spanish:

```
(cons (list "en") (list "fr" "es"))
```

A database is searched in a two-phase process. First, an appropriate callback is called to do the search: `define-word` is called for `DEFINE` searches and `match-word` is called for matches. This callback returns an opaque entity, called *result handle*, which is then passed to the `output` callback, which is responsible for outputting it.

`define-word` *dbh word* [Guile Callback]
Virtual Table: `(cons "define" define-word)`

Find definitions of word *word* in the database *dbh*. Return a result handle. If nothing is found, return `#f` or `'()`.

The argument *dbh* is the database handle returned by `open-db`.

`match-word` *dbh strat key* [Guile Callback]
Virtual Table: `(cons "match" match-word)`

Find in the database *dbh* all headwords that match *key*, using strategy *strat*. Return a result handle. If nothing is found, return `#f` or `'()`.

The *key* is a *Dico Key* object, which contains information about the word being looked for. To obtain the actual word, use the `dico-key->word` function (see [dico-key->word], page 53).

The argument *dbh* is a database handle returned by `open-db`. The matching strategy *strat* is a special Scheme object that can be accessed using a set of functions described below (see Section 5.5.4 [Dico Scheme Primitives], page 53).

`result-count` *resh* [Guile Callback]
Virtual Table: `(cons "result-count" result-count)`

Return the number of elements in the result set *resh*.

`output` *resh n* [Guile Callback]
Virtual Table: `(cons "output" output)`

Output *n*th result from the result set *resh*. The argument *resh* is a result handle returned by `define-word` or `match-word` callback.

The data must be output to the current output port, e.g. using `display` or `format` primitives. If *resh* represents a match result, the output must not be quoted or terminated by newlines.

It is guaranteed that the `output` callback will be called as many times as there are elements in *resh* (as determined by the `result-count` callback) and that for each subsequent call the value of *n* equals its value from the previous call incremented by one.

At the first call *n* equals 0.

5.5.4 Dico Scheme Primitives

GNU Dico provides the following Scheme primitives for accessing various fields of the `strat` and `key` arguments to `match` callback:

`dico-key?` *obj* [Function]
Return '`#t`' if *obj* is a Dico key object.

`dico-key->word` *key* [Function]
Extract the lookup word from the key object *key*.

`dico-make-key` *strat word* [Function]
Create new key object from strategy *strat* and word *word*.

`dico-strat-selector?` *strat* [Function]
Return true if *strat* has a selector (see Section 6.2.2 [Selector], page 74).

`dico-strat-select?` *strat word key* [Function]
Return true if *key* matches *word* as per strategy selector *strat*. The *key* is a '`Dico Key`' object.

`dico-strat-name` *strat* [Function]
Return the name of strategy *strat*.

`dico-strat-description` *strat* [Function]
Return a textual description of the strategy *strat*.

`dico-strat-default?` *strat* [Function]
Return `true` if *strat* is a default strategy. See Section B.2.2 [MATCH], page 104.

`dico-register-strat` *strat descr* [*fun*] [Function]
Register a new strategy. If *fun* is given it will be used as a callback for that strategy. Notice, that you can use strategies implemented in Guile in your C code as well (see Section B.2.2 [MATCH], page 104).

The selector function must be declared as follows:

```
(define (fun key word)
  ...)
```

It must return `#t` if *key* matches *word*, and `#f` otherwise.

5.5.5 Example Module

In this subsection we will show how to build a simple `dicod` module written in Scheme. The source code of this module, called `listdict.scm` and a short database for it, `numerals-pl.db`, are shipped with the distribution in the directory `examples`.

The database is stored in a disk file in form of a list. The first two elements of this list contain database description and full information strings. Rest of elements are conses, whose `car` contains the headword, and `cdr` contains the corresponding dictionary article. Following is an example of such a database:

```
("Short English-Norwegian numerals dictionary"
 "Short English-Norwegian dictionary of numerals (1 - 7)"
 ("one" . "en")
 ("two" . "to")
 ("three" . "tre")
 ("four" . "fire")
 ("five" . "fem")
 ("six" . "seks")
 ("seven" . "sju"))
```

We wish to declare such databases in `dicod.conf` the following way:

```
database {
        name "numerals";
        handler "guile example.db";
}
```

Thus, the `rest` argument to '`open-db`' callback will be '`("guile" "example.db")`' (see [open-db], page 50). Given this, we may write the callback as follows:

```
(define (open-db name . rest)
  (let ((db (with-input-from-file
                (cadr rest)
              (lambda () (read)))))
    (cond
     ((list? db) (cons name db))
     (else
      (format (current-error-port) "open-module: ~A: invalid format\n"
              (car args))
      #f))))
```

The list returned by this callback will then be passed as a database handle to another callback functions. To facilitate access to particular elements of this list, it is convenient to define the following syntax:

```
(define-syntax db:get
  (syntax-rules (info descr name corpus)
    ((db:get dbh name)   ;; Return the name of the database.
     (list-ref dbh 0))
    ((db:get dbh descr)  ;; Return the desctiption.
     (list-ref dbh 1))
    ((db:get dbh info)   ;; Return the info string.
```

```
      (list-ref dbh 2))
     ((db:get dbh corpus) ;; Return the word list.
      (list-tail dbh 3))))
```

Now, we can write '`descr`' and '`info`' callbacks:

```
(define (descr dbh)
  (db:get dbh descr))

(define (info dbh)
  (db:get dbh info))
```

The two callbacks '`define-word`' and '`match-word`' provide the core module functionality. Their results will be passed to '`output`' and '`result-count`' callbacks as a "result handler" argument. In the spirit of Scheme, we make the result a list. Its `car` is a boolean value: `#t`, if the result comes from '`define-word`' callback, and `#f` if it comes from '`match-word`'. The `cdr` of this list contains a list of matches. For '`define-word`', it is a list of conses copied from the database word list, whereas for '`match-word`', it is a list of headwords.

The '`define-word`' callback returns all list entries whose `car`s contain the look up word. It uses `mapcan` function, which is supposed to be defined elsewhere:

```
(define (define-word dbh word)
  (let ((res (mapcan (lambda (elt)
                       (and (string-ci=? word (car elt))
                            elt))
                     (db:get dbh corpus))))
    (and res (cons #t res))))
```

The '`match-word`' callback (see [match-word], page 52) takes three arguments: a database handler *dbh*, a strategy descriptor *strat*, and a word *word* to look for. The result handle it returns contains a list of headwords from the database that match *word* in the sense of *strat*. Thus, the behavior of '`match-word`' depends on the *strat*. To implement this, let's define a list of directly supported strategies (see below for definitions of particular '`match-`' functions):

```
(define strategy-list
  (list (cons "exact"  match-exact)
        (cons "prefix"  match-prefix)
        (cons "suffix"  match-suffix)))
```

The '`match-word`' callback will then select an entry from that list and call its `cdr`, e.g.:

```
(define (match-word dbh strat key)
  (let ((sp (assoc (dico-strat-name strat) strategy-list)))
    (let ((res (cond
                (sp
                 ((cdr sp) dbh strat (dico-key->word key)))
```

If the requested strategy is not in that list, the function will use the selector function if it is available, and the default matching function otherwise:

```
                ((dico-strat-selector? strat)
```

```
                 (match-selector dbh strat key))
                (else
                 (match-default dbh strat (dico-key->word key))))))
```

Notice the use of `dico-key->word` function to extract the actual lookup word from the key object.

To summarize, the '`match-word`' callback is:

```
(define (match-word dbh strat key)
  (let ((sp (assoc (dico-strat-name strat) strategy-list)))
    (let ((res (cond
                (sp
                 ((cdr sp) dbh strat (dico-key->word key)))
                ((dico-strat-selector? strat)
                 (match-selector dbh strat key))
                (else
                 (match-default dbh strat (dico-key->word key))))))
      (if res
          (cons #f res)
          #f))))
```

Now, let's create the '`match-`' functions it uses. The '`exact`' strategy is easy to implement:

```
(define (match-exact dbh strat word)
  (mapcan (lambda (elt)
            (and (string-ci=? word (car elt))
                 (car elt)))
          (db:get dbh corpus)))
```

The '`prefix`' and '`suffix`' strategies are implemented using SRFI-13 (see Section "SRFI-13" in *The Guile Reference Manual*) functions `string-prefix-ci?` and `string-suffix-ci?`, e.g.:

```
(define (match-prefix dbh strat word)
  (mapcan (lambda (elt)
            (and (string-prefix-ci? word (car elt))
                 (car elt)))
          (db:get dbh corpus)))
```

Notice that whereas the '`prefix`' strategy is defined by the server itself, the '`suffix`' strategy is an extension, and should therefore be registered:

```
(dico-register-strat "suffix" "Match word suffixes")
```

The `match-selector` function is pretty similar to its siblings, except that it uses `dico-strat-select?` (see Section 5.5.4 [Dico Scheme Primitives], page 53) to select the matching elements. This also leads to this function expecting a *key* as its third argument, in contrast to the previous matchers, which expect the actual lookup word there:

```
(define (match-selector dbh strat key)
  (mapcan (lambda (elt)
            (and (dico-strat-select? strat (car elt) key)
                 (car elt)))
          (db:get dbh corpus)))
```

Finally, the `match-default` is a variable that refers to the default matching strategy for this module, e.g.:

```
(define match-default match-prefix)
```

The two callbacks left to define are '`result-count`' and '`output`'. The first of them simply returns the number of elements in `cdr` of the result:

```
(define (result-count rh)
  (length (cdr rh)))
```

The behavior of '`output`' depends on whether the result is produced by '`define-word`' or by '`match-word`'.

```
(define (output rh n)
  (if (car rh)
      ;; Result comes from DEFINE command.
      (let ((res (list-ref (cdr rh) n)))
        (display (car res))
        (newline)
        (display (cdr res)))
      ;; Result comes from MATCH command.
      (display (list-ref (cdr rh) n))))
```

Finally, at the end of the module the callbacks are made known to `dicod` by the module initialization function:

```
(define-public (example-init arg)
  (list (cons "open" open-module)
        (cons "descr" descr)
        (cons "info" info)
        (cons "define" define-word)
        (cons "match" match-word)
        (cons "output" output)
        (cons "result-count" result-count)))
```

Notice, that in this implementation '`close-db`' callback was not needed.

5.6 Python

The `python` module provides an interface which allows programmers to write loadable modules in Python. The syntax for loading the module is:

```
load-module name {
  command "python"
          " init-script=name"
          " load-path=path"
          " root-class=name";
}
```

All parameters are optional:

`load-path=`*`path`* [python module]
Augments the default search path for Python modules. The format of *path* is the usual UNIX path specification: a colon-separated list of directory names.

`init-script=`*name* [python module]
Specifies the name of the initial Python source file. This file will be loaded and interpreted immediately after loading the module.

`root-class=`*name* [python module]
Sets the name of the Python root class, which is responsible for the dictionary operations.

A particular instance of the `python` module is loaded using the `handler` statement within a `database` block. This statement takes the same parameters as described above, plus any number of command line arguments, which will be passed to the root class constructor.

5.6.1 Python Dictionary Class

The dictionary class must define the following methods:

`__init__` *self *argv* [Method on `DictionaryClass`]
Class constructor. The *argv* array supplies positional arguments from the `handler` statement in the configuration file.

`open` *self dbname* [Method on `DictionaryClass`]
Opens the database named *dbname*. Returns '`True`' on success and '`False`' on failure.

`close` *self* [Method on `DictionaryClass`]
Closes the database.

`descr` *self* [Method on `DictionaryClass`]
Returns a short description of the database.

`info` *self* [Method on `DictionaryClass`]
Returns a text describing the database.

`lang` *self* [Method on `DictionaryClass`]
Optional. Returns supported languages as '`(`*src*`, `*dst*`)`'.

`define_word` *self word* [Method on `DictionaryClass`]
Defines *word*. Returns a result (an opaque Python object) if the definition was found or '`False`' otherwise.

`match_word` *self strat word* [Method on `DictionaryClass`]
Searches for *word* in the database using strategy *strat*. Returns a result (an opaque Python object) if some matches were found or '`False`' otherwise.

`output` *self result n* [Method on `DictionaryClass`]
Outputs *n*th result from the result set *result*.

`result_count` *self result* [Method on `DictionaryClass`]
Returns number of elements in the result set.

`compare_count` *self result* [Method on `DictionaryClass`]
Optional. Returns the number of comparisons performed when constructing the result set.

`result_headers` *self result hdr* [Method on `DictionaryClass`]
Optional. Returns a dictionary of MIME headers.

`free_result` *self result* [Method on `DictionaryClass`]
Reclaims any resources used by the result set.

5.6.2 Dico Python Primitives

`register_strat` *name descr* [*proc*] [Python primitive]
Registers new match strategy. The arguments are:

name Strategy name for use in the `MATCH` command.

descr The dscription, which will appear in the output of `SHOW STRAT` command.

proc Optional selector procedure.

If the *proc* argument is present, it must be the name of a Python function declared as:

```
def select(opcode key headword):
```

Its arguments are:

opcode Integer operation code.

key An `DicoSelectionKey` object identifying the search term (see Section 5.6.2.1 [DicoSelectionKey], page 60).

headword The headword being examined.

At the beginning of the search, the function is called with the '`DICO_SELECT_BEGIN`' as its *opcode* argument. It must perform the necessary initialization and return.

At the end of the search loop, the function is called with *opcode* '`DICO_SELECT_END`'. It must perform the necessary deinitialization procedures and exit.

In both cases, the *key* and *headword* arguments are not defined.

Within the search loop, the function will be called for each headword from the database. The *opcode* parameter will be '`DICO_SELECT_RUN`'. In this case the function must return '`True`' if the *headword* matches the *key* and '`False`' otherwise.

`register_markup` *name* [Python primitive]
Registers a markup *name*.

`current_markup` [Python primitive]
Returns the name of the current markup.

5.6.2.1 The `DicoSelectionKey` class

The `DicoSelectionKey` class represents a search key and is used when looking for matches. Calling `str` on the object of that class returns the search term itself, as does the `word` method:

`word` [Method on `DicoSelectionKey`]
Returns the search term. It is equivalent to the `__str__` attribute.

5.6.2.2 The `DicoStrategy` class

A match strategy is represented by an object of the `DicoStrategy` class.

`name` [Variable of `DicoStrategy`]
The name of that strategy.

`descr` [Variable of `DicoStrategy`]
Textual description of the strategy.

`has_selector` [Variable of `DicoStrategy`]
'`True`' if this strategy has a selector (see [Python Selector], page 59).

`name` *is_default* [Variable of `DicoStrategy`]
'`True`' if this is the default strategy.

`select` *headword key* [Method on `DicoStrategy`]
Returns '`True`' if *key* matches *headword* as per this strategy.

5.6.3 Python Example

In this subsection we will show a simple database module written in Python. This module handles simple textual databases in the following format:

- Empty lines and lines beginning with double dash are ignored.
- A line beginning with '`descr:`' introduces a short dictionary description for `SHOW DB`. The '`descr:`' prefix and the white space immediately following it are removed. E.g.:

```
descr: Short English-Norwegian numerals dictionary
```

- Lines beginning with '`info:`' provide a verbose description of the database. These lines are concatenated after removing the '`info:`' prefix and white space immediately following it. E.g.:

```
info: A short English-Norwegian (Bokmål) dictionary
info: of numerals.
info:
info: This dictionary is public domain.
```

- A line beginning with '`lang:`' defines source and destination languages for this dictionary. E.g.:

```
lang: en : nb
```

- Any line consisting of exactly two words defines a dictionary entry. E.g.:

```
one     en
two     to
three   tre
four    fire
```

Now, let's create a module for handling this format. First, we need to import Dico primitives (see Section 5.6.2 [Dico Python Primitives], page 59) and the 'sys' module. The latter is needed for output functions:

```
import dico
import sys
```

Then, a result class will be needed for match_word and define_word methods. It will contain the actual data in the variable 'result':

```
class DicoResult:
    # actual data.
    result = {}
    # number of comparisons.
    compcount = 0

    def __init__ (self, *argv):
        self.result = argv[0]
        if len (argv) == 2:
             self.compcount = argv[1]

    def count (self):
        return len (self.result)

    def output (self, n):
        pass

    def append (self, elt):
        self.result.append (elt)
```

The following two classes extend 'DicoResult' for use with 'DEFINE' and 'MATCH' operations. The define_word method will return an instance of the 'DicoDefineResult' class:

```
class DicoDefineResult (DicoResult):
    def output (self, n):
        print "%d. %s" % (n + 1, self.result[n])
        print "---------",
```

The match_word method will return an instance of the 'MatchResult' class:

```
class DicoMatchResult (DicoResult):
    def output (self, n):
        sys.stdout.softspace = 0
        print self.result[n],
```

Now, let's define the dictionary class:

```
class DicoModule:
```

```
        # The dictionary converted to associative array.
        adict =  {}
        # The database name.
        dbname = ''
        # The name of the corresponding disk file.
        filename = ''
        # A sort information about the database.
        mod_descr = ''
        # A verbose description of the database is kept.
        # as an array of strings.
        mod_info = []
        # A list of source and destination languages:
        langlist = ()
```

The class constructor takes a single argument, defining the name of the database file:

```
        def __init__ (self, *argv):
            self.filename = argv[0]
            pass
```

The ‘open’ method opens the database and reads its data:

```
        def open (self, dbname):
            self.dbname = dbname
            file = open (self.filename, "r")
            for line in file:
                if line.startswith ('--'):
                    continue
                if line.startswith ('descr: '):
                    self.mod_descr = line[7:].strip (' \n')
                    continue
                if line.startswith ('info: '):
                    self.mod_info.append (line[6:].strip (' \n'))
                    continue
                if line.startswith ('lang: '):
                    s = line[6:].strip (' \n').split(':', 2)
                    if (len(s) == 1):
                        self.langlist = (s[0].split (), \
                                         s[0].split ())
                    else:
                        self.langlist = (s[0].split (), \
                                         s[1].split ())
                    continue
                f = line.strip (' \n').split (' ', 1)
                if len (f) == 2:
                    self.adict[f[0].lower()] = f[1].strip (' ')
            file.close()
            return True
```

The database is kept entirely in memory, so there is no need for '`close`' method. However, it must be declared anyway:

```
        def close (self):
            return True
```

The methods returning database information are trivial:

```
        def descr (self):
            return self.mod_descr

        def info (self):
            return '\n'.join (self.mod_info)

        def lang (self):
            return self.langlist
```

The '`define_word`' method checks if the search term is present in the dictionary, and, if so, converts it to the `DicoDefineResult`:

```
        def define_word (self, word):
            if self.adict.has_key (word):
                return DicoDefineResult ([self.adict[word]])
            return False
```

The '`match_word`' method supports the '`exact`' strategy natively via the `has_key` attribute of `adict`:

```
        def match_word (self, strat, key):
            if strat.name == "exact":
                if self.adict.has_key (key.word.lower ()):
                    return DicoMatchResult \
                            ([self.adict[key.word.lower()]])
```

Other strategies are supported as long as they have selectors:

```
            elif strat.has_selector:
                res = DicoMatchResult ([], len (self.adict))
                for k in self.adict:
                    if strat.select (k, key):
                        res.append (k)
                if res.count > 0:
                    return res
            return False
```

The rest of methods rely on the result object to do the right thing:

```
        def output (self, rh, n):
            rh.output (n)
            return True

        def result_count (self, rh):
            return rh.count ()

        def compare_count (self, rh):
```

```
        return rh.compcount
```

5.7 Stratall

The `stratall` module provides a new strategy, called '`all`'. This strategy always returns a full list of headwords from the database, no matter what the actual search word is.

To load this strategy, use the following configuration statement:

```
load-module stratall;
```

Using this strategy on a full set of databases ('`MATCH * all ""`') produces enormous amount of output, which may induce a considerable strain on the server, therefore it is advised to block such usage as suggested in Section 4.3.13 [Strategies and Default Searches], page 32:

```
strategy all {
        deny-all yes;
}
```

5.8 Substr

The `substr` module provides a '`substr`' search strategy. This strategy matches a substring anywhere in the keyword. For example:

```
C: MATCH eng-deu substr orma
S: 152 207 matches found: list follows
S: eng-deu "abnormal"
S: eng-deu "conformable"
S: eng-deu "doorman"
S: eng-deu "format"
...
```

The loading procedure expects no arguments:

```
load-module substr;
```

5.9 Word

The `word` module provides the following strategies:

word	Match separate words within headwords.
first	Match the first word within headwords.
last	Match the last word within headwords.

The initialization procedure loads all three if given no arguments, as in

```
load-module word;
```

If arguments are given, the initialization procedure loads only those strategies that are listed in its command line. For example, the statement below loads only '`first`' and '`last`' strategies:

```
load-module word {
   command "word first last";
}
```

The following is an example of using one of those strategies in a dico session:

```
C: MATCH devdict word government
S: 152 1 matches found: list follows
S: devdict "MONARCHICAL GOVERNMENT"
S: .
S: 250 Command complete
```

5.10 Nprefix

The `nprefix` module provides a strategy similar to '`prefix`', but which returns the specified range of bytes. For example, the statement

`MATCH` *`dict`* `nprefix` *`skip#count#string`*

where *skip* and *count* are positive integer numbers, returns at most *count* headwords whose prefix matches *string*, omitting first *skip* unique matches.

The entire '*`skip#count#`*' construct is optional. If not supplied, the '`nprefix`' strategy behaves exactly as '`prefix`'.

The module is loaded using this simple statement:

```
load-module nprefix;
```

5.11 metaphone2

The `metaphone2` module provides a strategy based on *Double Metaphone* phonetic encoding algorithm, published by Lawrence Philips.

The module is normally loaded as follows:

```
load-module metaphone2;
```

The only available initialization parameter is

`size` *`number`* [metaphone2 parameter]

Defines the size of computed Double Metaphone codes, in characters. The default is 4.

```
load-module metaphone2 {
   command "metaphone2 size=16";
}
```

5.12 Pcre

The `pcre` module provides a matching strategy using Perl-compatible regular expressions. The module is loaded using a simple statement:

```
load-module pcre;
```

The strategy has the same name as the module and is reflected in the server's HELP output as shown below:

```
pcre  "Match using Perl-compatible regular expressions"
```

The headword argument to the `pcre` MATCH statement should be a valid Perl regular expression. It can optionally be enclosed in a pair of slashes, in which case one or more of the following flags can appear after the closing slash:

`a` The regexp is *anchored*, that is, it is constrained to match only at the first matching point in the string that is being searched.

`e` Ignore whitespace and '`#`' comments in the expression.

`i` Ignore case when matching.

`G` Inverts the *greediness* of the quantifiers so that they are not greedy by default, but become greedy if followed by '`?`'. The same can also be achieved by setting the '`(?U)`' option within the pattern.

Any of these flags can also be used in reverted case, which also reverts its meaning. For example, '`I`' means case-sensitive matching.

Here is an example of using this strategy in a dico session:

```
MATCH ! pcre "/\\stext/i"
```

5.13 ldap

The `ldap` module loads the support for LDAP user databases. It is available if Dico has been configured with LDAP.

The module needs no additional configuration parameters:

```
load-module ldap;
```

See Section 4.3.3.2 [ldap userdb], page 19, for a description of its use.

5.14 pam

The `pam` module implements user authentication via PAM. It can be used only with '`LOGIN`' and '`PLAIN`' GSASL authentication methods.

The module is loaded as follows:

```
load-module pam {
    command "pam [service=sname]";
}
```

where *sname* is the name of PAM service to use. If not supplied, '`dicod`' service will be used.

The user database is normally initialized as:

```
user-db "pam://localhost";
```

If `password-resource` statement is given, its value will be used as service name, instead of the one specified in the `load-module` statement, e.g.:

```
user-db "pam://localhost" {
    password-resource "local";
}
```

The `group-resource` statement is not used, because there is no mechanism to return textual data from PAM.

6 Dico Module Interface

This chapter describes the API for Dico loadable modules.

6.1 dico_database_module

Each module must export exactly one symbol of type `struct dico_database_module`. This symbol must be declared as

```
DICO_EXPORT(name, module)
```

where *name* is the name of the module file (without suffix). For example, a module `word.so` would have in its sourse the following declaration:

```
struct dico_database_module DICO_EXPORT(word, module) = {
...
};
```

The `dico_database_module` has the following members:

`unsigned dico_version` [dico_database_module]
Interface version being used. It is recommended to use the macro `DICO_MODULE_VERSION`, which keeps the version number of the current interface.

`unsigned dico_capabilities` [dico_database_module]
Module capabilities. As of version 2.4, this member can be one of the following:

`DICO_CAPA_DEFAULT`
: This module defines a handler for a specific database format.

`DICO_CAPA_NODB`
: This module does not handle any databases. When this capability is specified, `dicod` will call only the `dico_init` member of the structure.

  This capability is used by modules defining new matching strategies or authentication methods.

`int dico_init` (*int* `argc`, *char* `**argv`) [Dico Callback]
This callback is called right after loading the module. It is responsible for module initialization. The arguments are:

argc
: Number of elements in *argv*.

argv
: The command line given by `command` configuration statement (see Section 4.3.11 [Handlers], page 28), split into words. The element `argv[0]` is the name of the module. The element `argv[argc]` is 'NULL'. Word splitting follows the rules similar to those used in shell. In particular, a quoted string (using both single and double quotes) is handled as a single word.

If `dico_capabilities` is `DICO_CAPA_DEFAULT`, this method is optional. If `dico_capabilities` is set to `DICO_CAPA_NODB`, `dico_init` is mandatory and must be the only method defined.

`dico_handle_t dico_init_db` (*const char* ***db*, *int* ***argc***, *char* ****argv***) [Dico Callback]

Initialize the database. This method is called as a part of database initialization routine at startup of `dicod`, after processing `dictionary` configuration statement (see Section 4.3.12 [Databases], page 30). Its arguments are:

db The name of the database, as given by the `name` statement.

argc Number of elements in *argv*.

argv The command line given by `handler` configuration statement (see Section 4.3.12 [Databases], page 30). The array is '`NULL`'-terminated.

This method returns a *database handle*, an opaque structure identifying the database. This handle will be passed as the first argument to other methods. On error, `dico_init_db` shall return `NULL`.

Notice, that this function is not required to actually open the database, if the '`open`' notion is supported by the underlying mechanism. Another method, `dico_open` is responsible for that.

`int dico_free_db` (*dico_handle_t* ***dh***) [Dico Callback]

Reclaim any resources associated with database handle *dh*. This method is called as part of exit cleanup routine, before the main `dicod` process terminates.

It shall return '`0`' on success, or any non-'`0`' value on failure.

`int dico_open` (*dico_handle_t* ***dh***) [Dico Callback]

Open the database identified by the handle *dh*. This method is called as part of child process initialization routine.

It shall return '`0`' on success, or any non-'`0`' value on failure.

The `dico_open` method is optional.

`int dico_close` (*dico_handle_t* ***dh***) [Dico Callback]

Close the database identified by the handle *dh*. This method is called as part of child process termination routine.

It shall return '`0`' on success, or any non-'`0`' value on failure.

The `dico_close` method is optional, but if `dico_open` is defined, `dico_close` must be defined as well.

`char * dico_db_info` (*dico_handle_t* ***dh***) [Dico Callback]

Return a database information string for the database identified by *dh*. This function is called on each `SHOW INFO` command, unless an informational text for this database is supplied in the configuration file (see Section 4.3.12 [Databases], page 30). This value must be allocated using malloc(3). The caller is responsible for freeing it when no longer needed.

This method is optional.

`char * dico_db_descr` (*dico_handle_t* **dh**) [Dico Callback]
Return a short database description string for the database identified by *dh*. This function is called on each `SHOW DB` command, unless a description for this database is supplied in the configuration file (see Section 4.3.12 [Databases], page 30). This value must be allocated using malloc(3). The caller is responsible for freeing it when no longer needed.

This method is optional.

`dico_result_t dico_match` (*dico_handle_t* **dh**, [Dico Callback]
const dico_strategy_t **strat**, *const char* ***word**)
Use the strategy *strat* to search in the database *dh*, and return all headwords matching *word*.

This method returns a *result handle*, an opaque pointer that can then be used to display the obtained results. It returns `NULL` if no matches were found.

`dico_result_t dico_define` (*dico_handle_t* **dh**, [Dico Callback]
const char ***word**)
Find definitions of headword *word* in the database identified by *dh*.

This method returns a *result handle*, an opaque pointer that can then be used to display the obtained results. It returns `NULL` if no matches were found.

`int dico_output_result` (*dico_result_t* **rp**, *size_t* **n**, [Dico Callback]
dico_stream_t **str**)
The `dico_output_result` method outputs to stream *str* the *n*th result from result set *rp*. The latter is a result handle, obtained from a previous call to `dico_match` or `dico_define`.

Returns '`0`' on success, or any non-'`0`' value on failure.

It is guaranteed that the `dico_output_result` callback is called as many times as there are elements in *rp* (as determined by the `dico_result_count` callback, described below) and that for each subsequent call the value of *n* equals its value from the previous call incremented by one.

At the first call *n* equals 0.

`size_t dico_result_count` (*dico_result_t* **rp**) [Dico Callback]
Return the number of distinct elements in the result set identified by *rp*. The latter is a result handle, obtained from a previous call to `dico_match` or `dico_define`.

`size_t dico_compare_count` (*dico_result_t* **rp**) [Dico Callback]
Return the number of comparisons performed when constructing the result set identified by *rp*.

This method is optional.

`void dico_free_result` (*dico_result_t* **rp**) [Dico Callback]
Free any resources used by the result set *rp*, which is a result handle, obtained from a previous call to `dico_match` or `dico_define`.

`int dico_result_headers` (*dico_result_t* ***rp***, *dico_assoc_list_t* ***hdr***) [Dico Callback]
Populate associative list *hdr* with the headers describing result set *rp*. This callback is optional. If defined, it will be called before outputting the result set *rp* if `OPTION MIME` is in effect (see Section B.2.4 [OPTION], page 106).

`int dico_run_test` (*int* ***argc***, *char* ***argv***) [Dico Callback]
Runs unit tests for the module. Argument vector contains all command line arguments that follow the `--runtest` option, up to the '`--`' marker or end of line, whichever is encountered first.

6.2 Strategies

A search strategy is described by the following structure:

```
struct dico_strategy {
    char *name;           /* Strategy name */
    char *descr;          /* Strategy description */
    dico_select_t sel;    /* Selector function */
    void *closure;        /* Additional data for SEL */
    int is_default;       /* True, if this is a default strategy */
    dico_list_t stratcl; /* Strategy access control list */
};
```

The first two members are mandatory and must be defined for each strategy:

`char * name` [member of `struct dico_strategy`]
Short name of the strategy. It is used as second argument to the `MATCH` command (see Section B.2.2 [MATCH], page 104) and is displayed in the first column of output by the `SHOW STRAT` command (see Section B.2.3 [SHOW], page 105).

`char * descr` [member of `struct dico_strategy`]
Strategy description. It is the string shown in the second column of `SHOW STRAT` output (see Section B.2.3 [SHOW], page 105).

`dico_select_t sel` [member of `struct dico_strategy`]
A *selector function*, which is used in iterative matches to select matching headwords. The `sel` function is called for each headword in the database with the headword and search key as its arguments and returns 1 if the headword matches the key and 0 otherwise. The `dico_select_t` type is defined as:

```
typedef int (*dico_select_t) (int, dico_key_t,
                              const char *);
```

See Section 6.2.2 [Selector], page 74, for a detailed description.

`void * closure` [member of `struct dico_strategy`]
An opaque data pointer intended for use by the selector function.

`int is_default` [member of `struct dico_strategy`]
This member is set to 1 by the server if this strategy is selected as the default one (see [default strategy], page 4).

`dico_list_t stratcl` [member of `struct dico_strategy`]
A control list associated with this strategy. See Section 4.3.13 [Strategies and Default Searches], page 32.

6.2.1 Search Key Structure

The `dico_key_t` is defined as a pointer to the structure `dico_key`:

```
struct dico_key {
    char *word;
    void *call_data;
    dico_strategy_t strat;
    int flags;
};
```

The structure represents a search key for matching algorithms. Its members are:

`char * word` [member of `struct dico_key`]
The search word or expression.

`void * call_data` [member of `struct dico_key`]
A pointer to selector-specific data. If necessary, it can be initialized by the selector when called with the '`DICO_SELECT_BEGIN`' opcode and deallocated when called with the '`DICO_SELECT_END`' opcode.

`dico_strategy_t strat` [member of `struct dico_key`]
A pointer to the strategy structure.

`int flags` [member of `struct dico_key`]
Key-specific flags. These are used by the server.

The following functions are defined to operate on search keys:

`int dico_key_init` (*struct dico_key ***`key`**, dico_strategy_t* ***`strat`***, *const char **`word`*) [function]
Initialize the key structure *key* with the given strategy *strat* and search word *word*. If *strat* has a selector function, it will be called with the '`DICO_SELECT_BEGIN`' opcode (see Section 6.2.2 [Selector], page 74) to carry out the necessary initializations.

The *key* itself may point to any kind of memory storage.

`void dico_key_deinit` (*struct dico_key **`key`*) [function]
Deinitialize the `dico_key` structure initialized by a prior call to `dico_key_init`. If the key strategy has a selector, it will be called with the '`DICO_SELECT_END`' opcode.

Note that this function makes no assumptions about the storage type of *key*. If it points to a dynamically allocated memory, it is the caller responsibility to free it.

`int dico_key_match` (*struct dico_key* ***key***, *const char* ***word***) [function]

Match headword and key. Return 1 if they match, 0 if they don't match and -1 in case of error. This function calls the strategy selector with the '`DICO_SELECT_RUN`' opcode (see Section 6.2.2 [Selector], page 74). It is an error if the strategy selector is not defined.

6.2.2 Strategy Selectors

Wherever possible, modules should implement strategies using effective look up algorithms. For example, '`exact`' and '`prefix`' strategies must normally be implemented using binary search in the database index. The '`suffix`' strategy can also be implemented using binary search if a special *reverse index* is built for the database (this is the approach taken by `outline` and `dictorg` modules).

However, some strategies can only be implemented using a relatively expensive iteration over all keys in the database index. For example, '`soundex`' and '`levenshtein`' strategies cannot be implemented otherwise.

A strategy that can be used in iterative look ups must define a *selector*. Strategy selector is a function which is called for each database headword to determine whether it matches the search key.

It is defined as follows:

`int select` (*int* ***opcode***, *dico_key_t* ***key***, *const char* ***headword***) [selector]

A strategy selector. Its arguments are:

opcode The operation code. Its possible values are '`DICO_SELECT_BEGIN`', '`DICO_SELECT_RUN`' and '`DICO_SELECT_END`', as described below.

key The search key.

headword The database headword.

The selector function is called before entering the iteration loop with '`DICO_SELECT_BEGIN`' as its argument. If necessary, it can perform any additional initialization of the strategy, such as allocation of auxiliary data structures, etc. The `call_data` member of `dico_key_t` structure (see [dico_key], page 73) should be used to keep the pointer to the auxiliary data. The function should return 0 if it successfully finished its initialization and non-zero otherwise.

Once the iteration loop is finished, the selector will be called with '`DICO_SELECT_END`' as its first argument. This invocation is intended to

deallocate any auxiliary memory and release any additional resources allocated at the initialization state.

In these two additional invocations, the *headword* parameter will be 'NULL'.

Once the iteration loop is entered, the selector function will be called for each headword. Its *opcode* parameter will be 'DICO_SELECT_RUN' and the *headword* parameter will point to the headword. The function should return 1 if the headword matches the key, 0 if it does not and a negative value in case of failure.

To illustrate the concept of strategy selector, let's consider the implementation of the 'soundex' strategy in dicod. This strategy computes a four-character soundex code for both search key and the headword and returns 1 (match) if both codes coincide. To speed up the process, the code for the search key is computed only once, at the initialization stage, and stored in a temporary memory assigned to the key->call_data. This memory is reclaimed at the terminating call:

```
int
soundex_sel(int cmd, dico_key_t key, const char *dict_word)
{
    char dcode[DICO_SOUNDEX_SIZE];

    switch (cmd) {
    case DICO_SELECT_BEGIN:
        key->call_data = malloc(DICO_SOUNDEX_SIZE);
        if (!key->call_data)
            return 1;
        dico_soundex(key->word, key->call_data);
        break;

    case DICO_SELECT_RUN:
        dico_soundex(dict_word, dcode);
        return strcmp(dcode, key->call_data) == 0;

    case DICO_SELECT_END:
        free(key->call_data);
        break;
    }
    return 0;
}
```

6.3 Output

The dico_output_result method is called when the server needs to output the result of a 'define' or 'match' command. It must be defined as follows:

```
int output_result (dico_result_t rp, size_t n,
```

```
dico_stream_t str);
```

The *rp* argument points to the result in question. From the server's point of view it is an opaque pointer. The application shall define its own result structure, so normally the first operation the `dico_output_result` method does is typecasting *rp* to a pointer to that structure in order to be able to access its members.

A result can conceptually contain several *parts*. For example, the result of a '`DEFINE`' command can contain several definitions of the term. Similarly, the result of '`MATCH`' contains one or more matches. The server obtains the exact number of parts in a result by calling the `dico_result_count` method (see [dico_result_count], page 71).

When outputting a result, the server calls the `dico_output_result` in a loop, once for each result part. It passes the ordinal (zero-based) number of the part that needs to be output in the *n* parameter. It is guaranteed that *n* increases by one for each subsequent call of `dico_output_result` with the same *rp* parameter.

The *str* parameter identifies the *output stream*. The `dico_output_result` function must format the requested part from the result and output it to that stream. To do so it should use one of the following functions:

`int dico_stream_write` (*dico_stream_t* ***str***, *const void* ***buf***, *size_t* ***count***) [Function]

Writes *count* bytes from the buffer pointed to by *buf* to the output stream *str*. Returns 0 on success, and non-zero on error.

`int dico_stream_writeln` (*dico_stream_t* ***str***, *const char* ****buf***, *size_t* ***size***) [Function]

Same as `dico_stream_write`, but ends the output with a ***newline*** character (ASCII 10).

6.4 Module Unit Testing

The `dico_run_test` member of `struct dico_database_module` (see Section 6.1 [dico_database_module], page 69) points to the function that serves as entry point for unit tests of that module. If it is NULL, the module does not support unit testing. Otherwise, unit tests can be run using the following command line syntax:

```
$ dicod --runtest module [test_args] [-- init_args]
```

As usual, square brackets denote optional parts. The *module* argument specifies the name of the module to test. The arguments that follow the `--runtest` (`-r`) option are collected into two arrays: arguments up to the '`--`' marker form the vector that is passed to the module's `dico_run_test` function. The '`--`' marker is optional. If present, arguments that follow it are collected into a separate argument vector starting from slot 1, the slot 0 is set to point to the module name and the resulting vector is passed to the `dico_init` method of the module.

When running unit tests, configuration file is ignored. The diagnostic messages are printed to the standard error output.

Use the `--load-dir` (`-L`) command line option, if the module being tested cannot be found in the default load path (see [load path], page 29), e.g.:

```
$ dicod -L ../lib --runtest metaphone2 build A B C
```

7 Dico — a client program.

The `dico` program is a console-based utility for querying dictionary servers. It has two operation modes. In *single query mode*, the utility performs a query, displays its result and exits immediately. This mode is entered if a word or a URL was given in the command line. In *interactive mode*, the utility enters a read-and-eval loop, in which it reads requests from the keyboard, performs the necessary searches, and displays obtained results on the screen.

7.1 Single Query Mode

The simplest way to use `dico` utility is to invoke it with a word as an argument, e.g.:

```
$ dico entdeckung
```

In the example above, the utility will search definitions of the word '`entdeckung`' using its default server name and database. The default server name is read from the initialization file (see Section 7.3 [Initialization File], page 89). If it is not present, a predefined value specified at configuration time (see Section 3.2 [Default Server], page 7) is used. The default database is '`!`', which means "search in all available databases until a match is found, and then display all matches in that database".

There are two ways to change these defaults. First, you can use command line options. Secondly, you can use a DICT URL. Which method to use depends on your preferences. Both methods provide the same functionality for querying word definitions. However, command line options allow the user to query additional data from the server, which is impossible using URLs.

7.1.1 Dico Command Line Options

To connect to a particular dictionary server, use the `--host` option, for example:

```
$ dico --host dico.org entdeckung
```

To search in a particular database, use the `--database` (`-d`) option. For example, to display definitions from all databases:

```
$ dico --database '*' entdeckung
```

Note single quotes around the asterisk.

To get a list of databases offered by the server, use the `--dbs` (`-D`) option. In this case you may not give any non-option arguments. For example:

```
$ dico --dbs
```

If you wish to get a list of matches, instead of definitions, use the `--match` (`-m`) option. For example, the following invocation will display all matches from all the databases:

```
$ dico --database '*' --match entdeckung
```

The match mode uses '`.`' strategy by default (see Section B.2.2 [MATCH], page 104), which means a server-dependent default strategy, which suits best for interactive spell checking. To select another strategy, use the `--strategy` (`-s`) option.

If the remote server supports '`xlev`' experimental capability (see Section B.3 [Extended Commands], page 108, you may use the `--levdist` (`--levenshtein-distance`) option to set maximum Levenshtein distance, for example:

```
$ dico --levdist 2 --match entdeckung
```

Note that setting the distance too high is impractical and may imply unnecessary strain on the server.

To get a list of available matching strategies, with descriptions, use the `--strategies` (`-S`) option.

7.1.2 DICT URL

Another way to specify data for a query is by using URL, instead of a word to search, as in the example below:

```
$ dico dict://gnu.org.ua/d:entdeckung
```

A DICT URL consists of the following parts:

```
dict://user;pass@host:port/d:word:database:n
dict://user;pass@host:port/m:word:database:strat:n
```

The '`/d`' syntax requests the definition of *word*, whereas the '`/m`' syntax queries for matches, and is similar to the `--match` option. Some or all of '*`user;pass@`*', '*`:port`*', *database*, *strat*, and and *n* may be omitted. The meaning of all URL parts and their default values (if appropriate) are explained in the table below:

user — The user name to use in authentication. Similar to the `--user` option. If *user* is omitted and cannot be retrieved by other means, no authentication is attempted. See Section 7.4 [Autologin], page 90, for a detailed description of authentication procedure and sources which are used to obtain authentication credentials.

pass — A shared key (password) for that user. This part is similar to the `--key` command line option.

For compatibility with other URLs, `dico` tolerates a colon (instead of semicolon) as a delimiter between *user* and *pass*.

If *user* is given, but *pass* is not, `dico` will ask you to supply a password interactively (see Section 7.4 [Autologin], page 90).

host — Host name, IPv4 address, or IPv6 address (in square brackets) of the server to query. Same as the `--host` command line option.

port — Port number or service name (from `/etc/services`). If it is not present, the default of 2628 is used.

Same as the `--port` command line option.

word The word to look for.

database The database to search in. If not given, '`!`' is assumed.

Same as the `--database` command line option.

strat The matching strategy to use. If omitted, '`.`' is assumed.

Same as the `--strategy` command line option.

n Extract and display the *n*th definition of the word. If omitted, all definitions are displayed.

There is no command line option equivalent for this parameter, because it is used rarely.

Trailing colons may be omitted. For example, the following URLs might specify definitions or matches:

```
dict://dict.org/d:shortcake:
dict://dict.org/d:shortcake:*
dict://dict.org/d:shortcake:wordnet:
dict://dict.org/d:shortcake:wordnet:1
dict://dict.org/d:abcdefgh
dict://dict.org/d:sun
dict://dict.org/d:sun::1
dict://dict.org/m:sun
dict://dict.org/m:sun::soundex
dict://dict.org/m:sun:wordnet::1
dict://dict.org/m:sun::soundex:1
dict://dict.org/m:sun:::
```

7.2 Interactive Mode

If neither word nor URL nor any operation mode option were given on the command line, `dico` enters interactive mode. In this mode it reads commands from the standard input, executes them and displays results on the standard output. If the standard input is connected to a terminal, the readline and history facilities are enabled (see Section "Command Line Editing" in *GNU Readline Library*).

When in interactive mode, `dico` displays its prompt and waits for you to enter a command. The default prompt is the name of the program, followed by a '`greater than`' sign and a single space:

```
dico> _
```

The input syntax is designed so as to save you the maximum amount of typing.

If you type any word, the default action is to look up its definition using the default server and database settings, for example:

```
dico> man
From eng-swa, English-Swahili xFried/FreeDict Dictionary:
man  <n.>

    mwanamume
```

To match the word, instead of defining it, prefix it with a slash, much as you do in `vi`:

```
dico> /man
From eng-swa, English-Swahili xFried/FreeDict Dictionary:
0) ‘‘can’’
1) ‘‘man’’
2) ‘‘many’’
3) ‘‘map’’
4) ‘‘may’’
5) ‘‘men’’
```

Displayed is a list of matches retrieved using the default strategy. To see a definition for a particular match, type the number shown at its left. For example, to define “men”:

```
dico> 5
From eng-swa, English-Swahili xFried/FreeDict Dictionary:
men <n.>

    wanaume
```

Define and match are two basic actions. To discern from them, the rest of `dico` commands begin with a *command prefix*, a single punctuation character selected for this purpose. The default command prefix is a dot, but it can be changed using the `prefix` command (see Section 7.2.8 [Other Commands], page 87).

We will discuss the `dico` commands in the following subsections.

7.2.1 Server Commands

The `open` command establishes connection to a remote server. It takes up to two arguments, first of them specifying the IP or host name of the server, and the optional second one specifying the port number to connect to. For example:

```
dico> .open gnu.org.ua
```

If any or both of its arguments are absent, the `open` command reuses the value supplied with its previous invocation, or, if it is issued for the first time, the default values. The default for server name is ‘`gnu.org.ua`’ and the default port number is 2628. Both values can be changed at configuration time, see Section 3.2 [Default Server], page 7 for a detailed instruction.

When given one argument, `open` checks if it begins with a directory separator (‘`/`’). If so, the argument is handled as the full file name of a UNIX socket.

Note that you are not required to issue this command. If it is not given, `dico` will attempt to establish a connection using its default settings before executing any command that requires a connection to the server.

The `close` command closes the connection. It does not take any arguments.

7.2.2 Database and Strategy

The `database` command changes or displays the database name which is used by define and match commands. To display the database name, type the command without arguments:

```
dico> .database
!
```

To change the database, give its name as an argument to the command:

```
dico> .database *
```

Once the connection with the server is established, you may use command line completion facility to select the database from among those offered by the server. Typing *TAB* will show you a list of databases that begin with the characters you typed:

```
dico> .database enTAB
en-pl-naut  eng-afr     eng-deu     eng-swa
```

If you supply enough characters to identify a single choice, *TAB* will automatically select it. In the example above, typing a *TAB* after

```
dico> .database en-
```

completes the database name to:

```
dico> .database en-pl-naut
```

The `strategy` command displays or changes the default strategy name. As with `database`, the strategy completion is available for this command.

```
dico> .strategy
.
dico> .strategy dlev
```

If the remote server supports '`xlev`' experimental capability (see Section B.3 [Extended Commands], page 108), you can use the `distance` command to set the maximum Levenshtein distance for strategies that use Levenshtein algorithm. If used without arguments, this command displays the distance reported by the server and the configured distance, e.g.:

```
dico> .distance
Reported Levenshtein distance: 1
No distance configured
```

If used with a single numeric argument, it attempts to set the distance to the supplied value.

7.2.3 Informational Commands

The `ls` command lists available strategies (see Section B.2.3 [SHOW], page 105):

```
dico> .ls
exact "Match words exactly"
prefix "Match word prefixes"
soundex "Match using SOUNDEX algorithm"
all "Match everything (experimental)"
lev "Match headwords within given Levenshtein distance"
dlev "Match headwords within given Damerau-Levenshtein
      distance"
re "POSIX 1003.2 (modern) regular expressions"
regexp "Old (basic) regular expressions"
suffix "Match word suffixes"
rev-qu "Reverse search in Quechua databases"
```

The `ld` command lists available databases (see Section B.2.3 [SHOW], page 105):

```
dico> .ld
eng-swa "English-Swahili xFried/FreeDict Dictionary"
swa-eng "Swahili-English xFried/FreeDict Dictionary"
afr-eng "Afrikaans-English FreeDict Dictionary"
eng-afr "English-Afrikaans FreeDict Dictionary"
```

The `info` command displays information about a database, whose name is given as its argument. If used without arguments, it displays information about the current database.

```
dico> .info pl-en-naut
pl-en-naut - A Polish-English dictionary of nautical terms.
Copyright (C) 2008 Sergey Poznyakoff

Permission is granted to copy, distribute and/or modify this
document under the terms of the GNU Free Documentation
License, Version 1.2 or any later version published by the
Free Software Foundation; with no Invariant Sections, no
Front-Cover and Back-Cover Texts.
```

7.2.4 History Commands

Each issued command is stored in a history list and assigned a unique *event number*. When `dico` exits, it saves the command history to a file named `.dico_history` in your home directory. Upon startup, it retrieves the history from this file, so the history is preserved between sessions.

You can view the command history using the `history` command:

```
dico> .history
  1) .open dict.org
  2) entdeckung
```

```
3) /geschwindigkeit
```

A number of editing commands is provided, that allow you to refer to previous events from the history list and to edit them. For example, to reissue the 3rd event from the above list, type '`!3`'. The command with this index will be inserted at the `dico` prompt and you will be given a possibility to edit it. For a detailed description of all history-editing commands, please refer to Section "Using History Interactively" in *GNU History User Manual*.

7.2.5 Pager

When a command produces output that contains more lines than there are rows on the terminal, `dico` attempts to use a *pager program* to display it. The name (and arguments) of the pager program are taken from the `dico` internal variable, or, if it is not set, from the `PAGER` environment variable.

The `dico` pager setting can be examined or changed using the `pager` command. When used without arguments, it displays the current setting:

```
dico> .pager
less
(Pager set from environment)
```

When used with a single argument, it sets the pager:

```
dico> .pager "less -R"
```

The argument '`-`' (a dash) disables pager.

7.2.6 Program Settings

The commands described in this subsection are designed mostly for use in `dico` initialization file (see Section 7.3 [Initialization File], page 89).

The `autologin` command sets the name of autologin file to be used for authentication. When used without arguments, it displays the current setting. The argument to `autologin` command is subject to *tilde expansion*, i.e. if it begins with '`~/`', this prefix is replaced with the name of the current user home directory, followed by '`/`'. Similarly, a prefix '`~`*`login`*`/`' is replaced by the home directory for user *login*, followed by a slash.

See Section 7.4 [Autologin], page 90, for a detailed discussion of the autologin feature.

The `quiet` command toggles the display of `dico` startup banner. When started, `dico` prints a short list of information useful for beginning users: the program version and warranty conditions and a command to get help, e.g.:

```
dico 2.4
Copyright (C) 2005-2016 Sergey Poznyakoff
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Type ? for help summary

dico>
```

If you find this output superfluous and useless, you can suppress it by setting

```
quiet yes
```

in your initialization file.

7.2.7 Session Transcript

Session transcript is a special mode, which displays raw DICT commands and answers as they are executed. It is useful for debugging purposes.

You enable session transcript by issuing the following command:

```
dico> .transcript yes
# or
dico> .transcript on
```

Starting from then, each DICT transaction will be displayed on standard error output, for example:

```
dico> .open
dico: Debug: S:220 Pirx.gnu.org.ua dicod (dico 2.4)
  <mime.xversion.xlev> <32004.1216639476@gnu.org.ua>
dico: Debug: C:CLIENT "dico 1.99.91"
dico: Debug: S:250 ok
dico: Debug: C:SHOW DATABASES
dico: Debug: S:110 26 databases present
...
dico: Debug: S:.
dico: Debug: S:250 ok
dico: Debug: C:SHOW STRATEGIES
dico: Debug: S:111 10 strategies present: list follows
dico: Debug: S:exact "Match words exactly"
dico: Debug: S:prefix "Match word prefixes"
dico: Debug: S:soundex "Match using SOUNDEX algorithm"
...
dico: Debug: S:.
dico: Debug: S:250 ok
```

In the example above, ellipses are used to replace long lists of data. As you see, session transcripts may produce large amount of output.

To turn the session transcript off, use the following command:

```
dico> .transcript no
# or
dico> .transcript off
```

Finally, to query the current state of session transcript, issue this command without arguments:

```
dico> .transcript
transcript is on
```

7.2.8 Other Commands

The `prefix` command queries or changes the current command prefix:

```
dico> .prefix
Command prefix is .
dico> .prefix @
dico> @prefix
Command prefix is @
```

The `prompt` command changes the `dico` command line prompt. For example, to change it to '`dico$`', followed by a single space, type:

```
dico> .prompt "dico$ "
dico$ _
```

Note the use of quotes to include the space character in the argument.

The `help` command displays a short command usage summary. For convenience, a single question mark can be used instead of it:

```
dico> ?
/WORD                   Match WORD.
/                       Redisplay previous matches.
NUMBER                  Define NUMBERth match.
!NUMBER                 Edit NUMBERth previous command.

.open [HOST [PORT]]     Connect to a DICT server.
.close                  Close the connection.
...
```

The `version` command displays the package name and version number, and the `warranty` command displays the copyright statement.

Finally, the `quit` command leaves the dico shell. Typing end-of-file character (*C-d*) has the same effect.

7.2.9 Dico Command Summary

For convenience, this subsection lists all available `dico` commands along with their short description and a reference to the part of this manual where they are described in detail. The command names are given without prefix.

`open` *host port*
: Connect to a DICT server. Both arguments are optional. If any of them is absent, the value supplied with the previous `open` command is used. If there was no previous value, the default is used, i.e., '`gnu.org.ua`' for *host*, and 2628 for *port*.

See Section 7.2.1 [Server Commands], page 82.

`close`
: Close the connection.

See Section 7.2.1 [Server Commands], page 82.

`autologin` [*file*]
: Set or display the autologin file name.

See Section 7.4 [Autologin], page 90.

`sasl` [*bool*]
: Without argument, show whether the SASL authentication is enabled. With argument, enable or disable it, depending on the value of *bool*. See Section 7.4 [Autologin], page 90.

`database` [*name*]
: Set or display the current database name.

See Section 7.2.2 [Database and Strategy], page 83.

`strategy` [*name*]
: Set or display the current strategy name.

See Section 7.2.2 [Database and Strategy], page 83.

`distance` [*num*]
: Set or query Levenshtein distance. This command takes effect only if the remote server supports '`xlev`' experimental capability (see Section B.3 [Extended Commands], page 108).

See Section 7.2.2 [Database and Strategy], page 83.

`ls`
: List available matching strategies.

See Section 7.2.3 [Informational Commands], page 84.

`ld`
: List all accessible databases.

See Section 7.2.3 [Informational Commands], page 84.

`info` [*db*]
: Display information about the database *db*, or the current database, if used without argument.

`prefix` [*c*]
: Set or display the command prefix.

See Section 7.2.8 [Other Commands], page 87.

`transcript` [*bool*]
: Set or display session transcript mode.

See Section 7.2.7 [Session Transcript], page 86.

`verbose [number]`
: Set or display debugging verbosity level. Currently (as of version 2.4) it is a no-op.

`prompt string`
: Change command line prompt.

See Section 7.2.8 [Other Commands], page 87.

`pager string`
: Change or display pager settings.

See Section 7.2.5 [Pager], page 85.

`history`
: Display command history.

See Section 7.2.4 [History Commands], page 84.

`help`
: Display short command usage summary.

See Section 7.2.8 [Other Commands], page 87.

`version`
: Print program version.

See Section 7.2.8 [Other Commands], page 87.

`warranty`
: Print copyright statement.

See Section 7.2.8 [Other Commands], page 87.

`quiet bool`
: Toggle display of `dico` welcome banner. This command can be used only in initialization file.

See Section 7.2.6 [Program Settings], page 85.

`quit`
: Quit the shell.

See Section 7.2.8 [Other Commands], page 87.

7.3 Initialization File

When you start `dico`, it automatically executes commands from its *initialization files* (or *init files*, for short), normally called `.dico`. Two init files are read: the one located in your home directory, and the one from the current working directory. It is not an error if any or both of these files are absent.

These files contain a series of `dico` commands, as described in Section 7.2 [Interactive Mode], page 81, with the only difference that no command prefix is used by default. The '`#`' character introduces a comment: any characters from (and including) '`#`' up to the newline character are ignored[1].

Init files are useful to change the defaults for your `dico` invocation. Consider, for example, this init file:

[1] The same holds true for interactive mode as well, but you will hardly need comments on a terminal.

```
# An example init file for dico

# Turn the welcome banner off
quiet yes
# Set the location of autologin file
autologin ~/.dicologin
# Use this server by default
open dict.org
# Search in all databases
database *
# Finally, set the custom command prefix
prefix :
```

Notice, that if you wish to change your command prefix, it is preferable to do it as a last command in your init file, as shown in this example.

7.4 Autologin

After connecting to a remote server, `dico` checks if the server supports authentication and attempts to authenticate itself if so. To do this `dico` needs a set of parameters called *user credentials*. The exact set of credentials depends on the authentication mechanism being used, with user name and password being the two most important ones.

The user credentials can be supplied from the following sources:

1. Command line options `--user` and `--password`.
2. An URL given as a command line argument (see Section 7.1.2 [urls], page 80).
3. Autologin files.

These three sources are consulted in that order, i.e., a user name supplied with the `--user` command line option takes precedence over the one found in an URL and over any names supplied by autologin files.

If, after consulting all these sources, the user name is established, while the password is not, the resulting action depends on whether the standard input is connected to a terminal. If it is, `dico` will ask the user to supply a password. If it is not, authentication is aborted and connection to the server is closed.

Some authentication mechanisms require additional credentials. For example, GSSAPI authentication requires a *service name*. These credentials can be supplied only in autologin file.

Autologin file is a plaintext file that contains authentication information for various DICT servers. At most two autologin files are consulted: first the session-specific file, if it is supplied by `autologin` command (see Section 7.2.6 [Program Settings], page 85) or by the `--autologin` command line option, next the default file `.dicologin` in the user's home directory. The default

autologin file is examined only if no matching record was found in the session-specific one.

The autologin file format is similar to that of `.netrc` file used by `ftp` utility.

Comments are introduced by a pound sign. Anything starting from '`#`' up to the end of physical line is ignored.

Empty lines and comments are ignored.

Non-empty lines constitute *statements*. Tokens in a statement are separated with spaces, tabs, or newlines. A valid statement must begin with one of the following:

`machine` *name*
: This statement contains parameters for authenticating on machine *name*.

`default`
: This statement contains parameters for authenticating on any machine, except those explicitly listed in `machine` statements. There can be at most one `default` statement in autologin file. Its exact location does not matter, it will always be matched after all explicit `machine` statements.

During the lookup, `dico` searches the autologin file for a `machine` statement whose *name* matches the remote server name as given by `--host` command line option, host part of an URL (see Section 7.1.2 [urls], page 80), or the argument to the `open` command (see Section 7.2.1 [Server Commands], page 82). If it reaches end of the file without having found such an entry, it uses the `default` value, if available.

Once a matching entry is found, its subsequent tokens are analyzed. The following tokens are recognized:

`login` *name*
: Supply user name for this server.

`password` *string*
: Supply a password.

`noauth`
: Do not perform authentication on this machine.

`sasl`
: Enable SASL authentication.

`nosasl`
: Disable SASL authentication.

`mechanisms` *list*
: Declare acceptable SASL mechanisms. The *list* argument is a comma-separated list of mechanism names, without intervening whitespace. Multiple `mechanisms` may be given, in which case the corresponding lists are concatenated.

`service` *name*
: Declare service name, for authentication methods that need it. If this token is omitted, the default service name '`dico`' is used.

`realm` *name*
: Declare realm for authentication.

`host` *name*
: Set host name for this machine. By default, it is determined automatically.

Consider the following autologin entry, for example:

```
machine a.net user smith password guessme
machine b.net
  sasl
  mechanisms gssapi,digest-md5
  realm example.net
  service dico
  user smith password guessme
default noauth
```

When connecting to the server '`a.net`', `dico` will attempt the usual APOP authentication as user '`smith`' with password '`guessme`'. When connecting to the machine '`b.net`', it will use SASL authentication, via either GSSAPI or DIGEST-MD5 mechanisms, with realm name '`example.net`', service name '`dico`' and the same user name and password, as for '`a.net`'.

The authentication mechanism is suppressed if the `--noauth` option has been given in the command line, or a matching entry was found in one of the autologin files, which contained the `noauth` keyword.

7.5 Dico invocation

This section contains a short summary of `dico` command line options.

Command Line

The following table summarizes the four existing ways of `dico` invocation:

`dico [`*options*`]` *word*
: Connect to the dictionary and define or match a *word*.
See Section 7.1.1 [dico options], page 79.

`dico [`*options*`]` *url*
: Connect to the dictionary and define or match a word, supplied in the *url*.
See Section 7.1.2 [urls], page 80.

`dico [`*options*`]` *opmode*
: Connect to the dictionary and query the information required by *opmode* option, which is one of `--dbs`, `--strategies`, `--serverhelp`, `--info`, or `--serverinfo`. See below (see [dico-opmode], page 93) for a description.

`dico [`*options*`]`
: Start interactive shell. See Section 7.2 [Interactive Mode], page 81.

Server selection options:

`--host=`*server*
: Connect to this server.

  See Section 7.1.1 [dico options], page 79.

`--port=`*port*
`-p` *port*
: Specify the port to connect to. The argument *port* can be either a port number or its symbolic service name, as listed in `/etc/services`.

`--database=`*name*
`-d` *name*
: Select a database to search. The *name* can be either a name of one of the databases offered by the server (as returned by `--dbs` option), or one of the predefined database names: '`!`' or '`*`'.

  See Section 7.1.1 [dico options], page 79.

`--source=`*addr*
: Set source address for TCP connections.

Operation modifiers

`--match`
`-m`
: Match instead of define.

  See Section 7.1.1 [dico options], page 79.

`--strategy=`*name*
`-s` *name*
: Select a strategy for matching. The argument is either a name of one of the matching strategies supported by server (as displayed by `--strategies` option) or a dot ('`.`') meaning a server-dependent default strategy.

  This option implies `--match`.

  See Section 7.1.1 [dico options], page 79.

`--levdist=`*n*
`--levenshtein-distance=`*n*
: Sets maximum Levenshtein distance. Allowed values of *n* are between 1 and 9 inclusively. This option has effect only if the remote server supports '`xlev`' extension (see Section B.3 [Extended Commands], page 108).

  See Section 7.1.1 [dico options], page 79.

`--quiet`
`-q`
: Do not print the normal `dico` welcome banner when entering interactive shell.

  See Section 7.2.6 [Program Settings], page 85.

Operation modes

`--dbs`
`-D` Show available databases.

See Section 7.1.1 [dico options], page 79.

`--strategies`
`-S` Show available search strategies.

See Section 7.1.1 [dico options], page 79.

`--serverhelp`
`-H` Show server help.

`--info=`*`dbname`*
`-i` *`dbname`* Show information about database *dbname*.

`--serverinfo`
`-I` Show information about the server.

Authentication

`--noauth`
`-a` Disable authentication.

See Section 7.4 [Autologin], page 90.

`--sasl` Enable SASL authentication, if the server supports it. See Section 7.4 [Autologin], page 90.

`--nosasl` Disable SASL authentication. See Section 7.4 [Autologin], page 90.

`--user=`*`name`*
`-u` *`name`* Set user name for authentication.

See Section 7.4 [Autologin], page 90.

`--key=`*`string`*
`-k` *`string`*
`--password=`*`string`*
Set shared secret for authentication.

See Section 7.4 [Autologin], page 90.

`--autologin=`*`name`*
Set the name of autologin file to use.

See Section 7.4 [Autologin], page 90.

`--client=`*`string`*
`-c` *`string`* Additional text for client command, instead of the default '`GNU dico 2.4`'.

Debugging options

`--transcript`
`-t` Enable session transcript. See Section 7.2.7 [Session Transcript], page 86, for a description.

`--verbose`
`-v` Increase debugging verbosity level.

`--time-stamp`
Include time stamp in the debugging output.

`--source-info`
Include source line information in the debugging output.

Other options

`--help`
`-h` Display a short description of command line options.

`--usage` Display a short usage message

`--version`
Print program version.

8 GCIDER

`Gcider` is a window-based application for browsing the *GNU Collaborative International Dictionary of English* (GCIDE). When started, it launches a copy of `dicod` with a specially crafted configuration file and interfaces with it via stdin/stdout. For operation it needs to know the location of the `dicod` binary and of the directory where the GCIDE files reside. When started for the first time it will present you with a dialog box to help it locate the needed components. The location of the `dicod` binary is normally guessed by scanning the `PATH` environment variable. The only parameter you need to supply is the directory where the dictionary files reside. Once these data are entered, the program will save them in its configuration file (located in `~/.gcider`) and will reuse them in subsequent invocations.

The `gcider` user documentation is available online at http://dico.gnu.org.ua/gcider.html.

The program display is organized in three areas, ordered vertically. The topmost area is the *menu bar*, which contains pull-down menus. It is followed by a *search control* area. It provides an input line for you to enter the term to look-up in the dictionary, a set of widgets for bringing back prior inputs from the history and for controlling the search types and matching strategies. The area that follows presents two windows, side by side. The leftmost one is the *article window*, where definitions of the search terms are shown. The rightmost one is the *match list*, which will present the results of the recent *match* command. Finally, at the very bottom of the `gcider` window is located the *status bar*. Its purpose is twofold. First, it displays a status of the last search. Secondly, it provides a terse contextual help describing what you can do using the widget your mouse pointer points to.

To look up a word, type it in the input line in the search area and hit *CR* or click on the '`Define`' button. The definition, if found, is then displayed in the article window. This text may contain *cross-references* to other words in the dictionary, which are shown underlined, to draw your attention. To define a cross-reference, click on it with your mouse. You can also define any other word from the text. To do so, select it and click on the right button. Then, in the menu that will appear, select '`Define`'.

If you are not sure about the exact spelling of your search term, try searching for closest matches first. To do so, click on '`Match`' instead of '`Define`'. To find closest matches for a word in a definition, select the word (or part of it) and select '`Match`' in the contextual menu. In both cases, the program will try to match the word using the *strategy* selected currently in the strategy widget at the right of the search control area. Matching headwords will then be displayed in a listbox to the right of the article window. Clicking on a headword will bring its definition to the article window.

To select a match strategy, click on the strategy widget and select the desired strategy in the pop-down list that will appear. The list contains

short strategy names. To help you select the right one, the status line will show a full description of the currently highlighted strategy.

Those search terms for which a definition was found are saved in a *history list*. Several ways are provided to retrieve definitions from that list. First, clicking on the input widget brings a popdown list with all headwords from the history list shown in a reverse chronological order. Selecting a word from that list brings back its definition. Secondly, two special buttons to the right of the input widget can be used to navigate through the history. The button marked with a left arrow brings back previous definition, whereas the one marked with a right arrow brings back next definition.

By default the history list can accommodate up to 500 search terms. Once this limit reached, adding a new term to the list discards the oldest item, so that the total list length remains the same. Actual length of the history list can be configured using the `Edit/Appearance` menu.

9 How to Report a Bug

Email bug reports to bug-dico@gnu.org or bug-dico@gnu.org.ua. Please include a detailed description of the bug and information about the conditions under which it occurs, so we can reproduce it. To facilitate the task, the following list shows the basic set of information needed in order to find the bug:

- Package version you use. The output of `dicod --version` will do.
- A detailed description of the bug.
- Conditions under which the bug appears.
- It is often helpful to send the contents of `config.log` file along with your bug report. This file is created after running `./configure` in the source root directory of GNU Dico.

Appendix A Available Strategies

This appendix summarizes search strategies available for use in Dico 2.4.

exact — Match words exactly. This is a built-in strategy.

prefix — Match word prefixes. This is a built-in strategy.

nprefix — This strategy is similar to '`prefix`', except that it allows the user to limit the number of returned matches. If the search term has the structure '*`skip#count#string`*', where *skip* and *count* are integer numbers, then the '`nprefix`' strategy will return at most *count* headwords that begin with *string*, omitting first *skip* unique matches.

This strategy is implemented in the `nprefix` loadable module. See Section 5.10 [nprefix], page 65.

suffix — Match word suffixes. This is a built-in strategy.

soundex — Match words using SOUNDEX algorithm[1]. This strategy matches headwords that sound approximately the same as the search term. Note, that it is suitable only for English words.

This is a built-in strategy.

lev — Match headwords within given Levenshtein distance (1 by default). This strategy accounts for the most usual spelling errors.

The Levenshtein distance between two strings is the minimum number of *edits* needed to transform one string into the other. The edits are: insertion, deletion, or substitution of a single character. Thus, Levenshtein distance 1 means that only one such operation suffices to convert one string to another. This is the default for that strategy.

This built-in strategy is used as a default one (see [default strategy], page 4), unless the `default-strategy` configuration statement mandates otherwise.

The dictionary server may optionally allow users to alter the Levenshtein distance using the extension command `XLEV`. This command is enabled by setting the '`xlev`' capability. See Section 4.3.10 [Capabilities], page 28, for a detailed description.

nlev — Match normalized headwords within given Levenshtein distance. This strategy is similar to '`lev`', except that it treats any run-length of whitespace characters appearing in a headword as a single space (ASCII 32) character.

dlev — Match headwords within given Damerau-Levenshtein distance (1 by default).

[1] See `http://en.wikipedia.org/wiki/Soundex`

The Damerau-Levenshtein distance extends the Levenshtein distance by an additional edit operation: transposition of two adjacent characters.

This strategy is similar to '`lev`', but covers a much wider range of spelling and typographical errors.

The distance threshold optionally be configured using the `XLEV` command (see Section 4.3.10 [Capabilities], page 28).

ndlev — This is the same as '`dlev`', except that it treats any runlength of whitespace characters appearing in a headword as a single space (ASCII 32) character.

re — Match using POSIX 1003.2 regular expressions. This strategy treats the search term as a regular expression (see Section "Extended regular expressions" in *GNU sed*).

regexp — Match using basic regular expressions.

pcre — Match using Perl-compatible regular expressions. This strategy is implemented in the loadable module `pcre`. See Section 5.12 [pcre], page 65.

all — Match everything. This experimental strategy ignores its argument and matches all headwords. It is implemented in the `stratall` module, which you must load if you wish to make that strategy available. See Section 5.7 [stratall], page 64.

substr — Match a substring anywhere in the headword. This strategy is implemented as a loadable module. See Section 5.8 [substr], page 64.

word — Match a word anywhere in the headword. This is one of the strategies provided by the `word` loadable module. See Section 5.9 [word], page 64.

first — Match the first word within headwords. This strategy is implemented in `word` loadable module. See Section 5.9 [word], page 64.

last — Match the last word within headwords. This strategy is implemented in `word` loadable module. See Section 5.9 [word], page 64.

Appendix B Dictionary Server Protocol

This appendix describes commands understood by Dico dictionary server. The examples provided follow the convention used in RFC documents: a text sent by the server is prefixed with '`S`', whereas a text sent by the client is prefixed with '`C`'.

B.1 Initial Reply

When a connection is established, the server sends an *initial reply* to the client. This reply has the following format:

```
220 hostname text <capabilities> msg-id
```

Its parts and their meaning are described in the following table:

hostname The name of the host. It is determined automatically, unless set using `hostname` configuration file statement (see [hostname directive], page 27).

text Arbitrary text, as set via `initial-banner-text` configuration statement (see Section 4.3.9 [General Settings], page 26).

capabilities
A comma-separated list of server capabilities. It is configured using `capability` statement (see Section 4.3.10 [Capabilities], page 28).

msg-id A unique identifier similar to the format specified in RFC822, except that spaces and quoted pairs are not allowed within it.

This identifier will be used by the client when formulating the authentication string used in the `AUTH` command (see Section B.2.5 [AUTH], page 107).

An example of initial reply follows:

```
220 Trurl.gnu.org.ua <auth.mime> <520.1212912026@Trurl.gnu.org.ua>
```

B.2 Standard Commands

The following are standard commands, defined in RFC2229.

B.2.1 The DEFINE Command

The `DEFINE` command searches for definitions of a word.

`DEFINE` *db word* [Command]

Look up the word *word* in database *db*. If *db* is '`!`', then all the databases will be searched until the word is found, and all matches in that database will be returned. Similarly, if *db* is '`*`', then all the databases will be searched and all matches in all databases will be returned. In these two cases, the databases are searched in the same order as that returned by `SHOW DB` command (see Section B.2.3 [SHOW], page 105).

If the word was not found, response code 552 is returned.

If the word is found, a response code 150 is sent, followed by the number of definitions found. Then, for each definition a response code 151 is returned, followed by the textual body of the definition. In a 151 response, the first three space-delimited parameters give the word looked for, the name and a short description of the database. The latter two are the same as shown in the output from `SHOW DB` command.

The textual body of each definition is terminated with a dot ('`.`') on a line alone. If any line in the definition begins with a dot, it is duplicated to avoid confusion with body terminator.

After all of the definitions have been sent, a status code 250 is sent. If `timing` is set to '`true`' in the configuration file, this latter response also carries timing information. See Section 4.3.14 [Tuning], page 34, for more information about timing output.

Possible responses from `DEFINE` command are:

550 Invalid database, use SHOW DB for a list
552 No match
150 *n* definitions found: list follows
151 *word database name*
250 ok (optional timing information here)

Example transaction:

```
C: DEFINE eng-swa man
S: 150 1 definitions found: list follows
S: 151 "man" eng-swa "English-Swahili xFried/FreeDict Dictionary"
S: man  <n.>
S:
S:    mwanamume
S:
S: .
S: 250 Command complete [d/m/c = 1/0/12 0.000r 0.000u 0.000s]
```

B.2.2 The MATCH Command

The `MATCH` command searches for word in the database index. The searching algorithm is determined by *strategy*. See Section 6.2 [Strategies], page 72, for a list of strategies offered by GNU Dico.

`MATCH` *database strategy word* [Command]

Match *word* in *database* using *strategy*. As with `DEFINE`, the *database* can be '`!`' or '`*`' (See Section B.2.1 [DEFINE], page 103, for a detailed description of these wildcards).

The *strategy* is either the name of a strategy to use, or a dot ('`.`'), meaning to use default strategy. The default strategy is set using `default-strategy` configuration file statement (see Section 4.3.9 [General Settings], page 26. Its default value is '`lev`', which means '`use Levenshtein algorithm`' (see Section 6.2 [Strategies], page 72).

If no matches are found in any of the searched databases, then response code 552 will be returned. Otherwise, response code 152 will be returned followed by a list of matched words, one per line, in the form:

```
database word
```

Thus, prepending a '`DEFINE` ' to each such response, one obtains a valid `DEFINE` command.

The textual body of the match list is terminated with a line containing only a dot character.

Following the list, response code 250 is sent, which includes timing information, if `timing` directive is set in the configuration file (see Section 4.3.14 [Tuning], page 34).

Possible responses:

550 Invalid database, use SHOW DB for a list
551 Invalid strategy, use SHOW STRAT for a list
552 No match
152 *n* matches found: list follows
250 ok (optional timing information here)

Examples:

```
C: MATCH * . "weather"
S: 152 12 matches found: list follows
C: eng-afr "feather"
C: eng-afr "leather"
C: eng-afr "weather"
C: eng-deu "feather"
C: eng-deu "heather"
C: eng-deu "leather"
C: eng-deu "weather"
C: eng-deu "wether"
C: eng-deu "wheather"
C: devils "WEATHER"
S: .
S: 250 Command complete [d/m/c = 0/12/100677 0.489r 0.479u 0.007s]
```

B.2.3 The SHOW Command

The `SHOW` command outputs various information about the server and databases.

`SHOW DB` [Command]
`SHOW DATABASES` [Command]

Display the list of currently accessible databases, one per line, in the form:

```
database description
```

The list is terminated with is a dot ('`.`') on a line alone.

Possible responses:

110 *n* databases present
554 No databases present

`SHOW STRAT` [Command]
`SHOW STRATEGIES` [Command]
Display the list of currently supported search strategies, one per line, in the form:

```
strategy description
```

The list is terminated with is a dot (‘`.`’) on a line alone.

Possible responses:

111 *n* strategies available
555 No strategies available

`SHOW INFO` *database* [Command]
Displays the information about the specified database. The information is a free-form text and is suitable for display to the user in the same manner as a definition. The textual body of the response is terminated with is a dot (‘`.`’) on a line alone.

Possible responses:

550 Invalid database, use SHOW DB for a list
112 database information follows

The textual body is retrieved from the `info` statement in the configuration file (see Section 4.3.12 [Databases], page 30), or, if it is not specified, from the database itself, using `dico_db_info` callback function (see [dico_db_info], page 70). If neither source returns anything, the string ‘`No information available.`’ is returned.

`SHOW SERVER` [Command]
Return a server-specific information.

Response:

114 server information follows

The information follows, terminated with a dot on a line alone.

The textual body returned by the `SHOW SERVER` command consists of two parts. It begins with a line containing host name of the server and, optionally an additional information about the daemon and the system it runs on. The exact look and amount of information in this line is controlled by `show-sys-info` configuration statement (see Section 4.3.6 [Security Settings], page 23). This line is followed by the text supplied with `server-info` configuration statement (see Section 4.3.9 [General Settings], page 26).

B.2.4 The OPTION Command

The `OPTION` command allows to request optional features on the remote server. Currently the only implemented subcommand is:

`OPTION MIME` [Command]
Requests that all text responses be prefaced by a MIME header (RFC2045) followed by a single blank line.

After this command is issued, the server begins each textual response with a MIME header. This header consists of '`Content-type`' and '`Content-transfer-encoding`' headers, as supplied by the corresponding configuration file statements for this database (See Section 4.3.12 [Databases], page 30, see Section 4.3.12 [Databases], page 30). Any or both of these headers may be missing.

B.2.5 The AUTH Command

The `AUTH` command allows client to authenticate itself to the server. Depending on the server configuration, authenticated users may get access to more databases (see Section 4.3.12.1 [Database Visibility], page 32) or more detailed server information (see Section 4.3.6 [Security Settings], page 23).

`AUTH` *username auth-string* [Command]

Authenticate client to the server using a username and password. The *auth-string* is computed as in the APOP protocol (RFC1939. Briefly, it is the MD5 checksum of the concatenation of the *msg-id* (see Section B.1 [Initial Reply], page 103) and the *shared secret* that is stored both on the server and client machines.

See Section 4.3.3 [Authentication], page 16, for information on how to configure server for authenticating clients.

This command is supported only if '`auth`' capability is requested in the configuration (see Section 4.3.10 [Capabilities], page 28).

B.2.6 The CLIENT Command

`CLIENT` *info* [Command]

Identify client to server. The *info* argument contains a string identifying the client program (e.g. its name and version number). This information can then be used in logging (see Section 4.3.8 [Access Log], page 24).

B.2.7 The STATUS Command

`STATUS` [Command]

Display cumulative timing information. This command returns a '`210`' status code, followed by the timing information as described in Section 4.3.14 [Tuning], page 34, e.g.

```
C: STATUS
S: 210 [d/m/c = 28/1045/119856 21.180r 10.360u 1.040s]
```

B.2.8 The HELP Command

`HELP` [Command]

The `HELP` command provides a short summary of commands that are understood by the server. The response begins with a '`113`' code, followed by textual body defined in `help-text` configuration file statement (see

Section 4.3.9 [General Settings], page 26), which is terminated by a dot on a line by itself. A '250' response code finishes the output. For example:

```
113 help text follows
DEFINE database word         -- look up word in database
MATCH database strategy word -- match word in database
SHOW DB                      -- list all accessible databases
SHOW DATABASES               -- list all accessible databases
SHOW STRAT                   -- list available matching strategies
SHOW STRATEGIES              -- list available matching strategies
SHOW INFO database           -- provide database information
SHOW SERVER                  -- provide site-specific information
CLIENT info                  -- identify client to server
STATUS                       -- display timing information
HELP                         -- display this help information
QUIT                         -- terminate connection
.
250 Ok
```

B.2.9 The QUIT Command

`QUIT` [Command]

Terminate connection.

This command returns a response code 221, optionally followed by timing information (see Section 4.3.14 [Tuning], page 34).

B.3 Extended Commands

In addition to the standard commands, the Dico server also offers a set of experimental or extended commands.

`XIDLE` [Command]

This command displays the current inactivity timeout setting (see [inactivity-timeout], page 16), and resets idle timer to 0. The timeout value is printed as the first word after a '110' reply code, e.g.:

```
C: XIDLE
S: 110 180 second(s)
```

The value of '0' means there are no timeout.

`XVERSION` [Command]

This command displays the daemon implementation and version number. It becomes available only if '`xversion`' capability was requested in the configuration file (see Section 4.3.10 [Capabilities], page 28).

```
C: XVERSION
S: 110 dicod (dico 2.4)
```

`XLEV` *param* [Command]

If *param* is the word '`tell`', displays the current value of Levenshtein threshold. If *param* is a positive integer value, sets the Levenshtein threshold to this value.

This command becomes available only if '`xlev`' capability was requested in the configuration file (see Section 4.3.10 [Capabilities], page 28).

```
C: xlev tell
S: 280 1
C: xlev 3
S: 250 ok - Levenshtein threshold set to 3
C: xlev tell
S: 280 3
```

Appendix C Time and Date Formats

This appendix documents the time format specifications understood by the '`%t`' log format specifier (see Section 4.3.8 [Access Log], page 24). Essentially, it is a reproduction of the man page for GNU `strftime` function.

Ordinary characters placed in the format string are reproduced without conversion. Conversion specifiers are introduced by a '`%`' character, and are replaced as follows:

`%a` The abbreviated weekday name according to the current locale.

`%A` The full weekday name according to the current locale.

`%b` The abbreviated month name according to the current locale.

`%B` The full month name according to the current locale.

`%c` The preferred date and time representation for the current locale.

`%C` The century number (year/100) as a 2-digit integer.

`%d` The day of the month as a decimal number (range 01 to 31).

`%D` Equivalent to '`%m/%d/%y`'.

`%e` Like '`%d`', the day of the month as a decimal number, but a leading zero is replaced by a space.

`%E` Modifier: use alternative format, see below (see [conversion specs], page 113).

`%F` Equivalent to '`%Y-%m-%d`' (the ISO 8601 date format).

`%G` The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see '`%V`'). This has the same format and value as '`%y`', except that if the ISO week number belongs to the previous or next year, that year is used instead.

`%g` Like '`%G`', but without century, i.e., with a 2-digit year (00-99).

`%h` Equivalent to '`%b`'.

`%H` The hour as a decimal number using a 24-hour clock (range 00 to 23).

`%I` The hour as a decimal number using a 12-hour clock (range 01 to 12).

`%j` The day of the year as a decimal number (range 001 to 366).

`%k` The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also '`%H`'.)

`%l` The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also '`%I`'.)

`%m` The month as a decimal number (range 01 to 12).

`%M` The minute as a decimal number (range 00 to 59).

`%n` A newline character.

`%O` Modifier: use alternative format, see below (see [conversion specs], page 113).

`%p` Either '`AM`' or '`PM`' according to the given time value, or the corresponding strings for the current locale. Noon is treated as '`pm`' and midnight as '`am`'.

`%P` Like '`%p`' but in lowercase: '`am`' or '`pm`' or a corresponding string for the current locale.

`%r` The time in '`a.m.`' or '`p.m.`' notation. In the POSIX locale this is equivalent to '`%I:%M:%S %p`'.

`%R` The time in 24-hour notation ('`%H:%M`'). For a version including the seconds, see '`%T`' below.

`%s` The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.

`%S` The second as a decimal number (range 00 to 61).

`%t` A tab character.

`%T` The time in 24-hour notation (‘`%H:%M:%S`’).

`%u` The day of the week as a decimal, range 1 to 7, Monday being 1. See also ‘`%w`’.

`%U` The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also ‘`%V`’ and ‘`%W`’.

`%V` The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also ‘`%U`’ and ‘`%W`’.

`%w` The day of the week as a decimal, range 0 to 6, Sunday being 0. See also ‘`%u`’.

`%W` The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.

`%x` The preferred date representation for the current locale without the time.

`%X` The preferred time representation for the current locale without the date.

`%y` The year as a decimal number without a century (range 00 to 99).

`%Y` The year as a decimal number including the century.

`%z` The time-zone as hour offset from GMT. Required to emit RFC822-conformant dates (using ‘`%a, %d %b %Y %H:%M:%S %z`’)

`%Z` The time zone or name or abbreviation.

`%+` The date and time in *date(1)* format.

`%%` A literal ‘`%`’ character.

Some conversion specifiers can be modified by preceding them by the '`E`' or '`O`' modifier to indicate that an alternative format should be used. If the alternative format or specification does not exist for the current locale, the behaviour will be as if the unmodified conversion specification were used. The Single Unix Specification mentions '`%Ec`', '`%EC`', '`%Ex`', '`%EX`', '`%Ry`', '`%EY`', '`%Od`', '`%Oe`', '`%OH`', '`%OI`', '`%Om`', '`%OM`', '`%OS`', '`%Ou`', '`%OU`', '`%OV`', '`%Ow`', '`%OW`', '`%Oy`', where the effect of the '`O`' modifier is to use alternative numeric symbols (say, roman numerals), and that of the '`E`' modifier is to use a locale-dependent alternative representation.

Appendix D The Libdico Library

D.1 Strategies

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

```
struct dico_strategy {
    char *name;
    char *descr;
    dico_select_t sel;
    void *closure;
    int is_default;
};
```

`dico_strategy_t dico_strategy_dup` (*const dico_strategy_t* ***strat***) [Function]

`dico_strategy_t dico_strategy_find` (*const char* ***name***) [Function]

`int dico_strategy_add` (*const dico_strategy_t* ***strat***) [Function]

`dico_iterator_t dico_strategy_iterator` (*void*) [Function]

`void dico_strategy_iterate` (*dico_list_iterator_t* ***itr***, *void* ***data***) [Function]

`size_t dico_strategy_count` (*void*) [Function]

`int dico_set_default_strategy` (*const char* ***name***) [Function]

`const dico_strategy_t dico_get_default_strategy` (*void*) [Function]

`int dico_strategy_is_default_p` (*dico_strategy_t* ***strat***) [Function]

D.2 argcv

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

`dico_argcv_quoting_style` [enum]

`enum dico_argcv_quoting_style` [Variable]
`dico_argcv_quoting_style`

`int dico_argcv_get` (*const char ***command**, const char **delim*, const char **cmnt*, int **argc*, char ****argv*) [Function]

`int dico_argcv_get_n` (*const char ***command**, int **len**, const char **delim*, const char **cmnt*, int **argc*, char ****argv*) [Function]

`int dico_argcv_get_np` (*const char ***command**, int **len**, const char **delim*, const char **cmnt*, int **flags**, int **argc*, char ****argv*, char ***endp*) [Function]

`int dico_argcv_string` (*int **argc**, const char ***argv*, char ***string*) [Function]

`void dico_argcv_free` (*int **argc**, char ***argv*) [Function]

`void dico_argv_free` (*char ***argv*) [Function]

`int dico_argcv_unquote_char` (*int **c*) [Function]

`int dico_argcv_quote_char` (*int **c*) [Function]

`size_t dico_argcv_quoted_length` (*const char **str*, int **quote*) [Function]

`void dico_argcv_unquote_copy` (*char **dst*, const char **src*, size_t **n*) [Function]

`void dico_argcv_quote_copy` (*char **dst*, const char **src*) [Function]

`void dico_argcv_remove` (*int **argc*, char ****argv*, int (**sel*) (const char *, void *), void **data*) [Function]

D.3 Lists

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

`dico_list_t` [Type]

`dico_iterator_t` [Type]

`dico_list_iterator_t` [Function Type]

```
typedef int (*dico_list_iterator_t)(void *item, void *data);
```

`dico_list_comp_t` [Function Type]

```
typedef int (*dico_list_comp_t)(const void *, const void *);
```

`dico_list_t dico_list_create` (*void*) [Function]

`void dico_list_destroy` (*dico_list_t* ***list***, *dico_list_iterator_t* ***free***, *void* ***data***) [Function]

`void dico_list_iterate` (*dico_list_t* ***list***, *dico_list_iterator_t* ***itr***, *void* ***data***) [Function]

`void * dico_list_item` (*dico_list_t* ***list***, *size_t* ***n***) [Function]

`size_t dico_list_count` (*dico_list_t* ***list***) [Function]

`int dico_list_append` (*dico_list_t* ***list***, *void* ***data***) [Function]

`int dico_list_prepend` (*dico_list_t* ***list***, *void* ***data***) [Function]
`int dico_list_push` (*dico_list_t* ***list***, *void* ***data***) [Function]

`int dico_list_insert_sorted` (*dico_list_t* ***list***, *void* ***data***, *dico_list_comp_t* ***cmp***) [Function]

`dico_list_t dico_list_intersect` (*dico_list_t* **a**, *dico_list_t* **b**, *dico_list_comp_t* ***cmp***) [Function]

`int dico_list_intersect_p` (*dico_list_t* **a**, *dico_list_t* **b**, *dico_list_comp_t* ***cmp***) [Function]

`void * dico_list_pop` (*dico_list_t* ***list***) [Function]

`void * dico_list_locate` (*dico_list_t* ***list***, *void* ***data***, *dico_list_comp_t* ***cmp***) [Function]

`void * dico_list_remove` (*dico_list_t* ***list***, *void* ***data***, *dico_list_comp_t* ***cmp***) [Function]

`void * dico_iterator_current` (*dico_iterator_t* ***itr***) [Function]

`dico_iterator_t dico_iterator_create` (*dico_list_t* ***list***) [Function]

`void dico_iterator_destroy` (*dico_iterator_t* ***pitr***) [Function]

`void * dico_iterator_first` (*dico_iterator_t* ***itr***) [Function]

`void * dico_iterator_next` (*dico_iterator_t* ***itr***) [Function]

`void * dico_iterator_remove_current` (*dico_iterator_t* ***itr***) [Function]

`void dico_iterator_set_data` (*dico_iterator_t* ***itr***, *void* ***data***) [Function]

D.4 Associative lists

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

```
struct dico_assoc {
    char *key;
    char *value;
};
```

`dico_assoc_list_t` [Type]

`dico_assoc_list_t dico_assoc_create` (*void*) [Function]

`void dico_assoc_destroy` (*dico_assoc_list_t* ***passoc**) [Function]

`int dico_assoc_add` (*dico_assoc_list_t* **assoc**, *const char* ***key**, *const char* ***value**) [Function]

`const char * dico_assoc_find` (*dico_assoc_list_t* **assoc**, *const char* ***key**) [Function]

`void dico_assoc_remove` (*dico_assoc_list_t* **assoc**, *const char* ***key**) [Function]

D.5 Diagnostics Functions

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

L_DEBUG
L_INFO

L_NOTICE
L_WARN

L_ERR

L_CRIT

L_ALERT

L_EMERG

`const char * dico_program_name` [Variable]

`const char * dico_invocation_name` [Variable]

`void dico_set_program_name` (*char* ***name***) [Function]

`void dico_log_printer_t` (*int* ***lvl***, *int* ***exitcode***, *int* ***errcode***, *const char* ****fmt***, *va_list* ***ap***) [Function Type]

`void dico_set_log_printer` (*dico_log_printer_t* ***prt***) [Function]

`void dico_vlog` (*int* ***lvl***, *int* ***errcode***, *const char* ****fmt***, *va_list* ***ap***) [Function]

`void dico_log` (*int* ***lvl***, *int* ***errcode***, *const char* ****fmt***, ...) [Function]

`void dico_die` (*int* ***exitcode***, *int* ***lvl***, *int* ***errcode***, *char* ****fmt***, ...) [Function]

`int dico_str_to_diag_level` (*const char* ****str***) [Function]

`dico_stream_t dico_log_stream_create` (*int* ***level***) [Function]

D.6 Filter

`FILTER_ENCODE` [Define]

`FILTER_DECODE` [Define]

`filter_xcode_t` [Function Type]

```
typedef int (*filter_xcode_t) (const char *, size_t,
                               char *, size_t, size_t *, size_t, size_t *);
```

`dico_stream_t filter_stream_create` (*dico_stream_t* ***str***, *size_t* ***min_level***, *size_t* ***max_line_length***, *filter_xcode_t* ***xcode***, *int* ***mode***) [Function]

`dico_stream_t dico_codec_stream_create` (*const char* ****encoding***, *int* ***mode***, *dico_stream_t* ***transport***) [Function]

`dico_stream_t dico_base64_stream_create` (*dico_stream_t* ***str***, *int* ***mode***) [Function]

`dico_stream_t dico_qp_stream_create` (*dico_stream_t* ***str***, *int* ***mode***) [Function]

`int dico_base64_input` (*char* ***c***) [Function]

`int dico_base64_decode` (*const char* ****iptr***, *size_t* ***isize***, *char* ****optr***, *size_t* ***osize***, *size_t* ****pnbytes***, *size_t* ***line_max***, *size_t* ****pline_len***) [Function]

`int dico_base64_encode ( const char *iptr, size_t isize, char *optr, size_t osize, size_t *pnbytes, size_t line_max, size_t *pline_len)` [Function]

`int dico_qp_decode ( const char *iptr, size_t isize, char *optr, size_t osize, size_t *pnbytes, size_t line_max, size_t *pline_len)` [Function]

`int dico_qp_encode ( const char *iptr, size_t isize, char *optr, size_t osize, size_t *pnbytes, size_t line_max, size_t *pline_len)` [Function]

D.7 parseopt

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

`DICO_PARSEOPT_PARSE_ARGV0`
`DICO_PARSEOPT_PERMUTE`

`dico_opt_type` [Enumeration]

- `dico_opt_null`
- `dico_opt_bool`
- `dico_opt_bitmask`
- `dico_opt_bitmask_rev`
- `dico_opt_long`
- `dico_opt_string`
- `dico_opt_enum`
- `dico_opt_const`
- `dico_opt_const_string`

`dico_option` [struct]

```
struct dico_option {
    const char *name;
    size_t len;
    enum dico_opt_type type;
    void *data;
    union {
        long value;
        const char **enumstr;
    } v;
    int (*func) (struct dico_option *, const char *);
};
```

`DICO_OPTSTR` *name* [Macro]

`int dico_parseopt` (*struct dico_option* ***opt**, *int* **argc**, *char* ****argv**, *int* **flags**, *int* ***index**) [Function]

D.8 stream

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

`int dico_stream_create` (*dico_stream_t* ***pstream**, *int* **flags**, *void* ***data**) [Function]

`DICO_STREAM_READ`
`DICO_STREAM_WRITE`
`DICO_STREAM_SEEK`

`int dico_stream_open` (*dico_stream_t* **stream**) [Function]

`void dico_stream_set_open` (*dico_stream_t* **stream**, *int* (***openfn**) (*void *, int*)) [Function]

`void dico_stream_set_seek` (*dico_stream_t* **stream**, *int* (***fun_seek**) (*void *, off_t, int, off_t **)) [Function]

`void dico_stream_set_size` (*dico_stream_t* **stream**, *int* (***sizefn**) (*void *, off_t **)) [Function]

`void dico_stream_set_read` (*dico_stream_t* **stream**, *int* (***readfn**) (*void *, char *, size_t, size_t **)) [Function]

`void dico_stream_set_write` (*dico_stream_t* **stream**, *int* (***writefn**) (*void *, const char *, size_t, size_t **)) [Function]

`void dico_stream_set_flush` (*dico_stream_t* **stream**, *int* (***flushfn**) (*void **)) [Function]

`void dico_stream_set_close` (*dico_stream_t* **stream**, *int* (***closefn**) (*void **)) [Function]

`void dico_stream_set_destroy` (*dico_stream_t* **stream**, *int* (***destroyfn**) (*void **)) [Function]

`void dico_stream_set_ioctl` (*dico_stream_t* **stream**, *int* (***ctl**) (*void *, int, void **)) [Function]

`void dico_stream_set_error_string` (*dico_stream_t* **stream**, *const char **(***error_string**) (*void *, int*)) [Function]

`int dico_stream_set_buffer` (*dico_stream_t* **stream**, *enum dico_buffer_type* **type**, *size_t size*) [Function]

`dico_buffer_type` [Enumeration]

`dico_buffer_none`
`dico_buffer_line`
`dico_buffer_full`

`off_t dico_stream_seek` (*dico_stream_t* **`stream`**, *off_t* **`offset`**, *int* **`whence`**) [Function]

`DICO_SEEK_SET`
`DICO_SEEK_CUR`
`DICO_SEEK_END`

`int dico_stream_size` (*dico_stream_t* **`stream`**, *off_t* ***`psize`**) [Function]

`int dico_stream_read_unbuffered` (*dico_stream_t* **`stream`**, *void* ***`buf`**, *size_t* **`size`**, *size_t* ***`pread`**) [Function]

`int dico_stream_write_unbuffered` (*dico_stream_t* **`stream`**, *const void* ***`buf`**, *size_t* **`size`**, *size_t* ***`pwrite`**) [Function]

`int dico_stream_read` (*dico_stream_t* **`stream`**, *void* ***`buf`**, *size_t* **`size`**, *size_t* ***`pread`**) [Function]

`int dico_stream_readln` (*dico_stream_t* **`stream`**, *char* ***`buf`**, *size_t* **`size`**, *size_t* ***`pread`**) [Function]

`int dico_stream_getdelim` (*dico_stream_t* **`stream`**, *char* ****`pbuf`**, *size_t* ***`psize`**, *int* **`delim`**, *size_t* ***`pread`**) [Function]

`int dico_stream_getline` (*dico_stream_t* **`stream`**, *char* ****`pbuf`**, *size_t* ***`psize`**, *size_t* ***`pread`**) [Function]

`int dico_stream_write` (*dico_stream_t* **`stream`**, *const void* ***`buf`**, *size_t* **`size`**) [Function]

`int dico_stream_writeln` (*dico_stream_t* **`stream`**, *const char* ***`buf`**, *size_t* **`size`**) [Function]

`int dico_stream_ioctl` (*dico_stream_t* **`stream`**, *int* **`code`**, *void* ***`ptr`**) [Function]

`const char * dico_stream_strerror` (*dico_stream_t* **`stream`**, *int* **`rc`**) [Function]

`int dico_stream_last_error` (*dico_stream_t* **`stream`**) [Function]

`void dico_stream_clearerr` (*dico_stream_t* **`stream`**) [Function]

`int dico_stream_eof` (*dico_stream_t* **`stream`**) [Function]

`int dico_stream_flush` (*dico_stream_t* **`stream`**) [Function]

`int dico_stream_close` (*dico_stream_t* **`stream`**) [Function]

`void dico_stream_destroy` (*dico_stream_t* ***`stream`**) [Function]

`off_t dico_stream_bytes_in` (*dico_stream_t* **`stream`**) [Function]

`off_t dico_stream_bytes_out` (*dico_stream_t* **`stream`**) [Function]

D.9 url

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

`dico_url` [struct]

```
#define DICO_REQUEST_DEFINE 0
#define DICO_REQUEST_MATCH 1

struct dico_request {
    int type;
    char *word;
    char *database;
    char *strategy;
    unsigned long n;
};

struct dico_url {
    char *string;
    char *proto;
    char *host;
    int port;
    char *path;
    char *user;
    char *passwd;
    dico_assoc_list_t args;
    struct dico_request req;
};
```

`dico_url_t` [Pointer]

`int dico_url_parse` (*dico_url_t* ***purl***, *const char* ***str***) [Function]

`void dico_url_destroy` (*dico_url_t* ***purl***) [Function]

`const char * dico_url_get_arg` (*dico_url_t* ***url***, *const char* ***argname***) [Function]

`char * dico_url_full_path` (*dico_url_t* ***url***) [Function]

D.10 UTF-8

This section describes functions for handling UTF-8 strings. A UTF-8 character can be represented either as a multi-byte character or a wide character.

Multibyte character is a `char *` pointing to one or more bytes representing the UTF-8 character.

Wide character is an `unsigned` value identifying the character.

In the discussion below, a sequence of one or more multi-byte characters is called a *multi-byte string*. Multibyte strings terminate with a single '`nul`' (0) character.

A sequence of one or more wide characters is called a *wide character string*. Such strings terminate with a single 0 value.

D.10.1 Character sizes

`size_t utf8_char_width` (*const unsigned char *cp*) [Function]
Returns length in bytes of the UTF-8 character representation pointed to by *cp*.

`size_t utf8_strlen` (*const char *str*) [Function]
Returns number of UTF-8 characters (not bytes) in *str*.

`size_t utf8_wc_strlen` (*const unsigned *s*) [Function]
Returns number of wide characters in the wide character string *s*.

D.10.2 Iterating over UTF-8 strings

`utf8_iterator` [struct]
A data type for iterating over a string of UTF-8 characters. Defined as:

```
struct utf8_iterator {
    char *string;
    char *curptr;
    unsigned curwidth;
};
```

When iterating over characters in string, `curptr` points to the current character, and `curwidth` holds its length in bytes.

`int utf8_iter_isascii` (*struct utf8_iterator itr*) [Function]
Returns '`true`' if *itr* points to a ASCII character.

`int utf8_iter_end_p` (*struct utf8_iterator *itr*) [Function]
Returns '`true`' if *itr* reached end of the input string.

`int utf8_iter_first` (*struct utf8_iterator *itr*, [Function]
*unsigned char *str*)
Initializes *itr* for iterating over the string *str*. On success, positions *itr*`.curptr` to the next character from the input string, sets *itr*`.curwidth` to the length of that character in bytes, and returns '`0`'. If *str* is an empty string, returns '`1`'.

`int utf8_iter_next` (*struct utf8_iterator *itr*) [Function]
Positions *itr*`.curptr` to the next character from the input string. Sets *itr*`.curwidth` to the length of that character in bytes.

D.10.3 Conversions

The following functions convert between the two string representations.

`int utf8_mbtowc_internal` (*void *data*, *int* (**read*) (*void**), *unsigned int *pwc*) [Function]

Internal function for converting a single UTF-8 character to a corresponding wide character representation. The character to convert is obtained by calling the function pointed to by *read* with *data* as its only argument. If that call returns a non-positive value, the function sets `errno` to '`ENODATA`' and returns -1.

`int utf8_mbtowc` (*unsigned int *pwc*, *const char *r*, *size_t len*) [Function]

Converts first *len* characters from the multi-byte string *r* to wide character representation. On success, returns 0 and stores the result in *pwc*. The result pointer is allocated using `malloc`(3).

On error (invalid multi-byte sequence encountered), returns -1 and sets `errno` to '`EILSEQ`'.

`int utf8_wctomb` (*unsigned char *r*, *unsigned int* `wc`) [Function]

Stores the UTF-8 representation of the Unicode character wc in `r[0..5]`. Returns the number of bytes stored. If *wc* is out of range, return -1 and sets `errno` to '`EILSEQ`'.

`int utf8_wc_to_mbstr` (*const unsigned *word*, *size_t wordlen*, *char **retptr*) [Function]

Converts first *wordlen* characters of the wide character string *word* to multi-byte representation. The result is returned in *retptr*. It is allocated using `malloc`(3).

Returns 0 on success. On error, returns -1 and sets `errno` to one of the following values:

ENOMEM
: Not enough memory to allocate the return buffer.

EILSEQ
: An invalid wide character is encountered.

`int utf8_mbstr_to_wc` (*const char *str*, *unsigned **wptr*, *size_t *plen*) [Function]

Converts a multi-byte string from *str* to its wide character representation.

The result is returned in *retptr*. It is allocated using `malloc`(3).

Returns 0 on success. On error, returns -1 and sets `errno` to one of the following values:

ENOMEM
: Not enough memory to allocate the return buffer.

EILSEQ
: An invalid wide character is encountered.

`int utf8_mbstr_to_norm_wc` (*const char* ***str***, *unsigned* ***wptr***, *size_t* ***plen***) [Function]

Converts a multi-byte string from *str* to its wide character representation, replacing each run of one or more whitespace characters with a single space character (ASCII 32).

The result is returned in *retptr*. It is allocated using `malloc`(3).

Returns 0 on success. On error, returns -1 and sets `errno` to one of the following values:

ENOMEM
: Not enough memory to allocate the return buffer.

EILSEQ
: An invalid wide character is encountered.

D.10.4 Comparing UTF-8 strings

`int utf8_symcmp` (*unsigned char* ***a***, *unsigned char* ***b***) [Function]

Compares first UTF-8 characters from *a* and *b*.

`int utf8_symcasecmp` (*unsigned char* ***a***, *unsigned char* ***b***) [Function]

Compares first UTF-8 characters from *a* and *b*, ignoring the case.

`int utf8_strcasecmp` (*unsigned char* ***a***, *unsigned char* ***b***) [Function]

Compares the two UTF-8 strings *a* and *b*, ignoring the case of the characters.

`int utf8_strncasecmp` (*unsigned char* ***a***, *unsigned char* ***b***, *size_t* ***maxlen***) [Function]

Compares at most *maxlen* first characters from the two UTF-8 strings *a* and *b*, ignoring the case of the characters.

`int utf8_wc_strcmp` (*const unsigned* ***a***, *const unsigned* ***b***) [Function]

Compare the two wide character strings *a* and *b*.

`int utf8_wc_strncmp` (*const unsigned* ***a***, *const unsigned* ***b***, *size_t* ***n***) [Function]

Compares at most *n* first characters from the wide character strings *a* and *b*.

`int utf8_wc_strcasecmp` (*const unsigned* ***a***, *const unsigned* ***b***) [Function]

Compares the two wide character strings *a* and *b*, ignoring the case of the characters.

`int utf8_wc_strncasecmp` (*const unsigned* ***a***, *const unsigned* ***b***, *size_t* ***n***) [Function]

Compares at most first *n* characters of the two wide character strings *a* and *b*, ignoring the case.

D.10.5 Character lookups

`unsigned * utf8_wc_strchr` (*const unsigned* ***str***, *unsigned* ***chr***) [Function]
Returns a pointer to the first occurrence of wide character *chr* in string *str*, or 'NULL', if no such character is encountered.

`unsigned * utf8_wc_strchr_ci` (*const unsigned* ***str***, *unsigned* ***chr***) [Function]
Returns a pointer to the first occurrence of wide character *chr* (case-insensitive) in string *str*, or 'NULL', if no such character is encountered.

`const unsigned * utf8_wc_strstr` (*const unsigned* ****vartext***, *const unsigned* ****pattern***) [Function]
Finds the first occurrence of *pattern* in *text*. Returns a pointer to the beginning of pattern in *text*. Returns NULL if no occurrence was found.

D.10.6 Functions for converting UTF-8 characters

`unsigned utf8_wc_toupper` (*unsigned* ***wc***) [Function]
Converts wide character *wc* to upper case, if possible. Returns *wc*, if it cannot be converted.

`int utf8_toupper` (*char* ****s***, *size_t* ***len***) [Function]
Converts first *len* bytes of the UTF-8 string *s* to upper case, if possible.

`unsigned utf8_wc_tolower` (*unsigned* ***wc***) [Function]
Converts wide character *wc* to lower case, if possible. Returns *wc*, if it cannot be converted.

`int utf8_tolower` (*char* ****s***, *size_t* ***len***) [Function]
Converts first *len* bytes of the UTF-8 string *s* to lower case, if possible.

`void utf8_wc_strupper` (*unsigned* ****str***) [Function]
Converts each character from the wide character string *str* to uppercase, if applicable.

`void utf8_wc_strlower` (*unsigned* ****str***) [Function]
Converts each character from the wide character string *str* to lowercase, if applicable.

D.10.7 Additional functions

`unsigned * utf8_wc_strdup` (*const unsigned* ****s***) [Function]
Returns a pointer to a new wide character string which is a duplicate of the string *s*. Memory for the new string is obtained with `malloc`(3), and can be freed with `free`(3).

`unsigned * utf8_wc_quote` (*const unsigned *s*) [Function]
Quotes occurrences of backslash and double-quote in *s* by prefixing each of them with a backslash. The return value is allocated using `malloc`(3).

`int utf8_quote` (*const char *str, char **sptr*) [Function]
Quotes occurrences of backslash and double-quote in *s* by prefixing each of them with a backslash. On success stores the result (allocated with `malloc`(3)) in *sptr*, and returns 0. On error, returns -1 and sets `errno` to the one of the following:

ENOMEM
: Not enough memory to allocate the return buffer.

EILSEQ
: An invalid wide character is encountered.

`size_t utf8_wc_hash_string` (*const unsigned *ws, size_t n*) [Function]
Compute a hash code of *ws* for a symbol table of *n* buckets.

`int dico_levenshtein_distance` (*const char *a, const char *b, int flags*) [Function]
Computes Levenshtein distance between UTF-8 strings *a* and *b*. The *flags* argument is a bitwise or of one or more flags:

`0`
: Default - compute Levenstein distance, treating both arguments literally.

`DICO_LEV_NORM`
: Treat runs of one or more whitespace characters as a single space character (ASCII 32).

`DICO_LEV_DAMERAU`
: Compute Damerau-Levenshtein distance. This distance takes into account transpositions.

`int dico_soundex` (*const char *word, char code*[*DICO_SOUNDEX_SIZE*]) [Function]
Computes the Soundex code for the given *word*. The code is stored in *code*. Returns 0 on success, -1 if *word* is not a valid UTF-8 string.

`DICO_SOUNDEX_SIZE` [Define]
This macro definition expands to the size of Soundex code buffer, including the terminal zero.

Note that this function silently ignores all characters, except Latin letters.

D.11 util

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

`char * dico_full_file_name` (*const char *`dir`*, *const char *`file`*) [Function]

`size_t dico_trim_nl` (*char *`buf`*) [Function]

`size_t dico_trim_ws` (*char *`buf`*) [Function]

D.12 xlat

==========

Editor's note:

The information in this node may be obsolete or otherwise inaccurate. This message will disappear, once this node revised.

==========

`xlat_tab` [struct]

```
struct xlat_tab {
    char *string;
    int num;
};
```

`int xlat_string` (*struct xlat_tab *`tab`*, *const char *`string`*, *size_t `len`*, *int `flags`*, *int *`result`*) [Function]

`int xlat_c_string` (*struct xlat_tab *`tab`*, *const char *`string`*, *int `flags`*, *int *`result`*); [Function]

XLAT_ICASE

Appendix E GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or

to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not

add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new

versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year your name*.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ‘‘GNU Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

> with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

This is a general index of all issues discussed in this manual.

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X