CLICON Tutorial

Olof Hagsand and Benny Holmgren

Jan, 2015

1 Introduction

CLICON is a software suite for configuration management including CLI generation, netconf interface and embedded databases. White-box systems, embedded devices and other systems can with a few steps add a conistent management interface with CLI, netconf access, realtime-database and transactions support.

This tutorial gives an overview of a simple application for configuring NTP, the Network Time Protocol. The complete application contains a CLI, an embedded database, a configuration engine and a Netconf interface.

The application is a part of the CLICON source-code release and can be accessed in the directory `appdir/ntp`. You can install the tutorial example after installing CLICON as follows:

```
> cd appdir ntp
> make
> sudo make install
```

Thereafter, `clicon_backend` and `clicon_cli` are started.

2 Application flow and commit semantics

Figure 1 shows a typical CLICON system where a user interacts with a local CLI to configure and manage a system. In the small example of this tutorial, the user manages an NTP service. The user adds new NTP servers, configures logging and shows the status of the service.

Example:

```
olof@ntp> ntp server ?
  <ipv4addr>              IPv4 address of peer
olof@ntp> ntp server 1.2.3.4
olof@ntp> ntp logging status on
olof@ntp> show configuration
ntp server 1.2.3.4
ntp logging status true
olof@ntp>quit
```

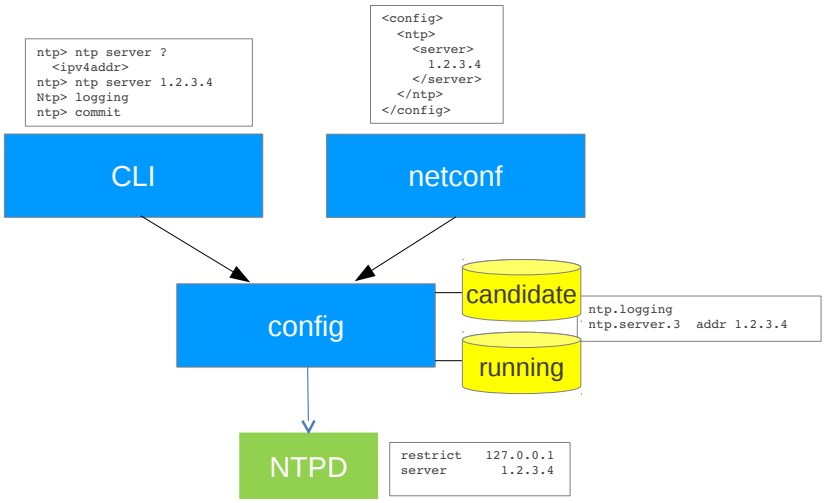

Figure 1: *An example of application flow in CLICON. Users enter commands as CLI or XML statements. A backend daemon manages the candidate and running databases and updates the system (such as the NTP daemon) when database entries change.*

The CLI statements above result in a change in the database which in turn rewrites the NTP configuration file. The new statements take effect when the NTP daemon is notified of the new configuration. The corresponding database and ntpd statements can be seen in Figure 1.

Looking in some more detail, the CLI statements are sent as modification statements to a *backend* daemon which adds them in a *candidate* database. When the user *commits* the change, the config daemon copies the new state into a *running* database, changes the NTP configuration file and restarts the NTP daemon.

The backend daemon knows how to update the NTP configuration file since there are associations between database entries and callbacks. When, for example, an `ntp.server` entry is added in the running database, a `server` entry is added to the NTP configuration file, and the daemon is restarted.

If the commit fails, the whole transaction is reverted, the running database is rolled back to a previous state, and the user is notified of the failed commit.

Note that as an alternative, one can define the CLI to use *auto-commit* which means that every configuration statement is executed directly and feedback on failure will be immediate. The ntp example uses autocommit.

The commit semantics as described follows Netconf [4]. One can use a Netconf client to modify or access configured data using XML. The netconf statement in Figure 1 shows a subset of an `edit-config` command, which accomplished an equivalent modification as the CLI command.

The CLI typically has more functionality than handling configurations. For example, it may have commands for examining the configured state, for invoking operations (eg 'ping'), and for showing statistics. However, it is the handling of configurations that is the main objective of the CLICON software, and thus of this tutorial.

3 Software architecture and plugins

CLICON provides a runtime with configuration interfaces for embedded applications. An application developer designs new applications by editing specification files and programming plugin C-code.

When done, the application is "clicked-on" offering an interactive CLI, a configuration daemon with transaction semantics, an embedded database and a netconf interface for remote machine-machine configuration.

Figure 2 shows a CLICON runtime with a CLI client, a Netconf client and a backend daemon. The figure shows an extended example, not only NTP, but also a hello functionality. The clients communicate with the backend server over a Unix domain socket. A configuration file contains common configure options for the system as a whole.

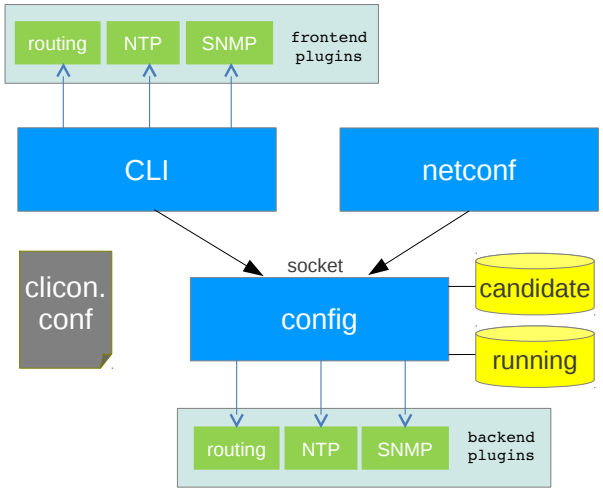

Figure 2: *CLI and Netconf clients communicate with the backend server over a Unix socket. The CLI loads frontend plugins, and the config daemon loads backend plugins.*

An important concept in CLICON is the use of application-specific *frontend* and *backend plugins*.

3.1 Frontend plugins: CLI and netconf

Frontend plugins, as shown in the upper part of Figure 2, define how users enter commands in the CLI or process specific messages in Netconf. Netconf plugins are not always necessary, since the standard functionality is often enough.

CLI plugins contains user callbacks that define how CLI commands are translated into the underlying database. Many of these commands are pre-definied by the system, for a simple system, the CLI plugin can be quite small. The CLI client loads the frontend plugins at runtime into its executable image and loads CLI specifications, defines callbacks, etc. In the figure, there are two CLI and two netconf plugins: NTP and hello.

3.2 Backend plugins

Application-specific backend plugins are loaded by the configuration daemon. Each backend-plugin *registers* callbacks with database entries: When the entries change, the callbacks are called. The callbacks performs some action depending on the new settings of the database entries.

Figure 2 shows two backend plugins: NTP and hello. The NTP plugin configures the NTP configuration files and restarts the NTPD daemon. The hello backend is just an example.

4 Developing applications

CLICON in itself consists of client and daemon binaries, libraries and include files. A developer designs an application by editing the *specification files* and programming the *plugins*.

Figure 3 shows the developing environment for the NTP application[1]. The figure shows the following specification files:

- clicon.conf - Basic CLICON configuration file, see Appendix B for more information. Basic and global configuration options are defined here.
- ntp.spec - The configuration database describes the configuration data, and what state is stored in the database. It may also be used to generate CLI syntax. See Section 5 for detailed information.
- ntp_cli.cli - The syntax specification of the CLI commands. Section 6 shows the syntax specification of the NTP application.
- ntp_cli.c - Contains callbacks for the CLI commands. This is described in more detail in Section 7.

[1]The figure is not strictly correct since it mixes target binaries with source code.

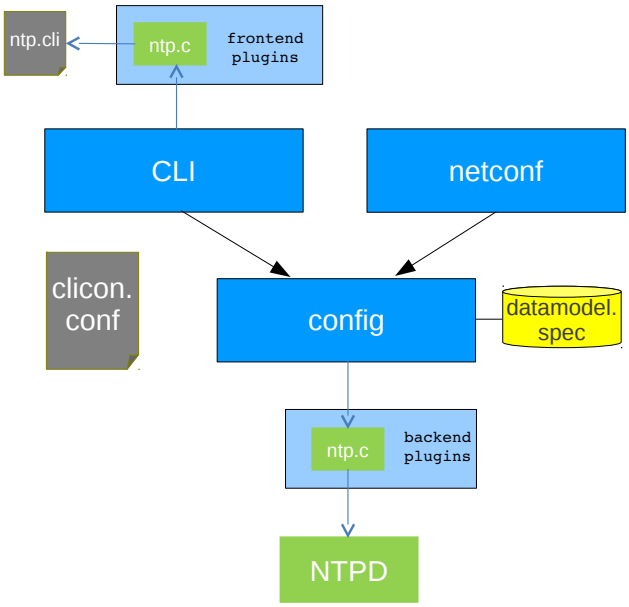

Figure 3: *Example of CLICON with plugin source code, CLI syntax and data-model specifications.*

- ntp_backend.c - The backend plugin registers functional callbacks, triggered when database entries change. The backend module of the NTP application is described in more detail in Section 8.
- ntp_netconf.c - The netconf plugin defines netconf commands that are non-standard. Typically operational commands.

5 The database specification: ntp.spec

The data model specifies a set of keys and values of keys used in the embedded database. The syntax of the data model is a tree-formed specification similar to the CLIgen specification syntax [1] also used in Section 6.

The data model for the NTP example consists of IP addresses to NTP servers and a logging flag:

```
ntp[ipv4addr]{                  # List
  server <ipv4addr:ipv4addr>;  # index variable (key)
  logging <status:bool default:off>;  # Single entry
}
```

The data model as shown above contains a variable assignment and two kinds of configuration entries: a list of servers each containing a unique IPv4 address, and a single status value on whether to perform logging or not. The exclamation mark in front of the address shows that the servers is a set and that the address is the unique index variable of that set.

In the example, there is only one variable per entry, but a data model can have several variables per entry.

The following example shows a somewhat more complex specification:

```
a[x]{
  <x:int>;
  <y:string>;
  <z:int default:(int)42>;
  <p:string mandatory>;
  b <q[]:int>;
}
```

The specification above allows a list of `a` entries where `x` is a key (index) variable being unique for all entries. In the example `y` and `q` are optional, `p` is mandatory and `z` has the default value `42` if not given.

Note that mandatory attributes must be given for the validation check to succeed. Example:

```
> a 23
> validate
Config error: key a: Missing mandatory attribute: p
> a 23 p foo
> a 23 b q 30
> a 23 b q 31
```

```
> show config
a {
    x 23;
    p foo;
    z 42;
    b {
        q 31;
        q 30;
    }
}
```

Note that (1) as long as `z` is not explicitly given, its value is set to the default value `42`. (2) that the `p` attribute must be given to pass the validation check and (3) the list of `q` entries specified by `q[]`.

The data model specification is central in CLICON for several reasons:

1. It determines which database entries are valid,
2. CLI configuration commands may be generated from the data model.
3. It determines valid netconf XML configuration statements.

Based on the data model, the runtime (embedded) databases (eg running and candidate in Figure 1) will be populated by database keys as specified by the model.

Note that the data model describes a tree, while the database consist of keys. This mapping is straightforward and is illustrated as follows [2]

```
ntp.server.3  ipv4addr=1.2.3.4
ntp.server.2  ipv4addr=4.45.5.6
ntp.server.0  ipv4addr=2.2.2.2
ntp.logging   status=true
```

The listing shows the content of a runtime database. The three `ntp.server` keys have one variable each, the index variable `ipv4addr` defining the unique index. The single ntp.logging key containing a single variable that will be overwritten if a new value is given.

6 The CLI syntax specification: ntp_cli.cli

The CLI syntax is specified in a syntax definition language as specified in CLIgen [1]. The CLI specification is a tree of commands with associated help texts.

The complete CLI syntax for the NTP application is as follows[3]:

```
CLICON_MODE="ntp";
CLICON_PLUGIN="ntp_cli";
CLICON_PROMPT="%U@%H> ";
show{
```

[2] The output is a summarized from the `clicon_dbctrl` application

[3] Help texts have been removed for clarity

```
    associations, cli_run("ntpq -p");
    configuration, show_conf_as_cli("running ^.*$");
    netconf, show_conf_as_netconf("running ^.*$");
}
@SYNTAX, cli_merge();
no @SYNTAX, cli_del();
quit, cli_quit();
```

Note the following:

The specification starts with variables defining the CLI behaviour. Appendix C lists such variables with their default behaviour.

Each leaf in the tree may specify a callback function with arguments. For example, "show associations" invokes the `cli_run` command with `ntpq -p` as argument.

All callbacks in this example (such as `cli_run`) are standard callbacks defined in the CLICON library. Many useful callbacks included the ones in the example are listed in Appendix D.

However, if the developer needs to define application-specific callbacks, this is done in the CLI plugin, in this case `ntp_cli.c` as defined by the `CLICON_PLUGIN` variable. How to define such a callbacks is described in CLIgen documentation [1].

The CLI syntax also has two example of *tree reference* in the form of the '@SYNTAX'. This means that a macro expansion is being made, substituting the reference to the whole data model as specified in Section 5.

For example, the line `no @SYNTAX, cli_del();` is translated to:

```
no ntp{
   server <ipv4addr:ipv4addr>, cli_del();
   logging status <status:bool>, cli_del();
}
```

where also comments and variables are substituted. The advantage with this is that configuration statements need not be entered again in the CLI specification, thus making the specification smaller.

If the application needs better control of the CLI configuration syntax, different from the generated, it is also possible to manually define how CLI commands are translated into database keys using explicit calls to `cli_set, cli_merge and cli_del`.

6.1 CLIGEN variables

You need to set some cligen variables to control the frontend:

- CLICON_PROMPT. Prompt to use in this mode. using %H (host) %U (user) and %T (terminal) format.
- CLICON_PLUGIN. Which frontend plugin (.so) to lookup callback functions in. You can use callback functions in the cligen syntax only from

one plugin, or from the clicon library. If you do not specify a plugin, the MASTER_PLUGIN will be used.

- CLICON_MODE. name of this mode. Use with -m option and when changing mode.

7 Frontend plugin: ntp_cli.c

The CLI plugin contains user-defined callbacks and may make system initialization programmatically. The NTP example needs no such specializations.

For CLI plugins, there are also some predefined callbacks available, the most common are:

- `plugin_init` - called as soon as the plugin has been loaded and is assumed initialize the plugin's internal state.
- `plugin_start` - is called once everything has been initialized, right before the main event loop is entered.
- `plugin_exit` - called in each plugin when the application is about to exit. It gives each plugin the chance to clean up after itself and do general housekeeping.
- `plugin_prompt_hook` - Before each CLI prompt is printed, this function is called giving the applications the chance to return a custom prompt.

8 Backend plugin: ntp_backend.c

The backend plugin consists of a predefined functions that follow a naming standard allowing them to be called from the configuration daemon. The initiator callbacks also typically sets up validation and commit callbacks.

Once the plugin is setup, the backend daemon reacts on transactions from the candidate to running database when a commit statement has been made. In case of autocommit, every configuration statement results in a transaction.

A transaction is a set of configuration commands that are executed as a whole. If one configuration statement fails the transaction is aborted and rollbacked to previous state.

- `plugin_init` is the first function called and sets up the dependency between database symbols and callbacks. In this case, any time an NTP entry changes, the `ntp_commit` function is called. The latter function actually only sets a variable for later processing. `plugin_start` - is called once everything has been initialized, right before the main event loop is entered.
- `transaction_begin` is called as the first callback in a transaction after a commit or validate command.

- `validation callbacks` as defined in `plugin_init` are called for every database entry that has changed. A validation callback should check the validity of the candidate database.
- `transaction_complete` is called when validation of all entries is complete.
- `commit callbacks` as defined in `plugin_init` are called for every database entry that has changed. The commit callbacks may change the actual system state.
- `transaction_end` is called after all commit functions are called. In this example, the actual communication with the NTP daemon is done here. If this would fail, the commit itself would be cancelled. `transaction_abort` is called if a commit calback fails and after the rollback.
- `plugin_exit` - called in each plugin when the application is about to exit. It gives each plugin the chance to clean up after itself and do general housekeeping.

8.1 Init

A backend plugin registers database keys. When the values of these keys change, the plugin gets notified. so that changes to these can be processed by the plugin. This is strictly only necessary in the incremental node.

Key registration is made using the `dpdep` function. The following example shows an init function registrating the keys "ntp.server[]" and "ntp.logging"[4]:

```
int
plugin_init(config_handle h)
{
   dbdep(h, 0, ntp_commit, NULL, 2, "ntp.server[]");
   dbdep(h, 0, ntp_commit, NULL, 2, "ntp.logging");
}
```

In the NTP example however, the small-granular validation and commit is not used. Instead, the complete configuration update is made in `transaction_end`, and NTPD is restarted.

Other applications may need to make incremental changes to the state, and then the commit callbacks are useful.

References

[1] Olof Hagsand, *CLIgen tutorial*, CLIgen technical documentation, April, 2011

[2] Olof Hagsand and Benny Holmgren, "CLICON data model", Technical documentation, September 2013.

[3] Olof Hagsand and Benny Holmgren, "CLICON db2txt fromatting", Technical documentation, September 2013.

[4] The actual code uses a somewhat more general approach

[4] R. Enns, Ed, *NETCONF Configuration protocol*, RFC 4741, IETF, Dec, 2006

Appendix A: Installation

CLICON can be installed on a variety of platforms using configure, then compiled and installed. Installation installs libraries, binaries and default configuration files in the system. It is also possible to install a development environment using install-include.

Note that CLIgen needs to be installed on your system before CLICON can be installed:

```
> git clone https://github.com/olofhagsand/cligen.git
> cd cligen
> ./configure
> make
> sudo make install
```

The, a typical CLICON installation is as follows:

```
> configure         # Configure clicon to platform
> make              # Compile
> sudo make install # Install libs, binaries, and config-files
> sudo make install-include # Install include files (for compiling)
```

To build and install the example "hello" and "ntp" applications:

```
> cd appdir
> make
> sudo make install
```

Note that ntp is installed as default application. To override this and make hello the default application, do as follows:

```
> cd appdir/hello
> make
> sudo make install
```

To run the default application (eg ntp):

```
> sudo clicon_config
clicon_config[17651]: clicon_config: 17651 Started
> clicon_cli
olof@ntp>
```

The following files are installed under the root installation directory.

```
bin:
   clicon_cli     # CLI frontend application
   clicon_dbctrl  # Database controller application
   clicon_netconf # Netconf frontend application
etc:
   cliconrc       # Source script file. Run to setup shell if
                  # installation is in non-standard place.
lib:
   libclicon_backend.so  # Lib for backend
```

```
   libclicon_cli.so      # Lib for cli frontend
   libclicon.so          # Lib for basic clicon
   libclicon_netconf.so  # Lib for netconf frontend
sbin:
   clicon_backend # Clicon backend application (run as daemon)
share/clicon:     # Default application directory including example
   clicon.conf    # Application configuration file (default ntp)
   hello.spec     # Data model for hello example app
   ntp.spec       # Data model for ntp example app
share/clicon/frontend:
   hello_cli.so      # Hello frontend cli plugin
   hello_cli.cli     # Hello cli syntax
   ntp_cli.so        # NTP frontend cli plugin
   ntp_cli.cli       # NTP  cli syntax
share/clicon/backend:
   hello_backend.so  # hello backend plugin
   ntp_backend.so    # NTP backend plugin
share/clicon/netconf:
   hello_netconf.so  # Hello netconf plugin
   ntp_netconf.so    # NTP netconf plugin
```

Some configure options are:

- `--prefix` Change root installation directory. Deafult is `/usr/local`.
- `--with-appdir` Set application directory, ie, where default clicon application directories should be installed. Default is `$prefix/share/clicon`.

Non-standard installation

If CLICON is installed in a non-standard installation, binaries and dynamic library needs to be identified. For this an rc file is provided in the etc directory.

Example:

```
> ./configure --prefix=/hello
> make
> make install
> make install-include
> cd <somewhere else>
> source /hello/etc/cliconrc
> clicon_config
> clicon_cli
```

Appendix B: CLICON Options

There are many CLICON options to control the installation and behaviour of CLICON.

Options with default values are given in the clicon configuration file (`clicon.conf`). Some option values can be overriden by command-line options when starting a clicon binary (eg clicon_cli or clicon_backend).

The CLICON_APPDIR option in particular can also be specified when configuring CLICON or as an environment variable.

A developer can also add application-specific options and use in the plugins.

The CLICON system options are as follows:

Name	Default value	Description
CLICON_APPDIR	/usr/local/share/clicon	Application installation directory.
CLICON_CONFIGFILE	$APPDIR/clicon.conf	File containing default option values.
CLICON_DBSPEC_TYPE	PT	PT (new) or KEY (old syntax)
CLICON_DBSPEC_FILE	$APPDIR/datamodel.spec	Location of data model
CLICON_CANDIDATE_DB	$APPDIR/db/candidate_db	Location of runtime candidate database
CLICON_RUNNING_DB	$APPDIR/db/running_db	Location of runtime running database
CLICON_ARCHIVE_DIR	$APPDIR/archive	Archive dir for checkpoints
CLICON_STARTUP_CONFIG	$APPDIR/startup-config	Startup config-file
CLICON_SOCK	$APPDIR/clicon.sock	Unix domain socket
CLICON_SOCK_GROUP	clicon	Unix group for clicon socket group access
CLICON_AUTOCOMMIT	0	Automatically commit configuration changes (no commit)
CLICON_QUIET		Do not print greetings on stdout. Eg clicon_cli -q
CLICON_MASTER_PLUGIN	master	Master plugin name. backend and CLI
CLICON_BACKEND_DIR	$APPDIR/backend/¡group¿	Dirs of all backend plugins
CLICON_BACKEND_PIDFILE	$APPDIR/clicon.pidfile	Process id file of clicon_backend daemon
CLICON_CLI_DIR	$APPDIR/frontend	Dir of all CLI plugins/ syntax group dirs
CLICON_CLI_GROUP		cli syntax group to start in (clicon_cli)
CLICON_CLI_MODE	base	Initial cli syntax mode to start in clicon_cli.
CLICON_CLI_GENMODEL	OFF	name of generated tree or OFF
CLICON_CLI_GENMODEL_TYPE	VARS	How to generate CLI from model: VARS or ALL
CLICON_CLI_GENMODEL_COMPLETION	0	Generate completion-rules for all variables.
CLICON_CLI_GENMODEL_OPTIONAL	0	If 0, all model variables are mandatory, if 1 all are optional
CLICON_CLI_COMMENT	#	comment char in CLI.
CLICON_CLI_VARONLY	1	Dont include keys in cvec in cli vars callbacks. New apps should use 0.
CLICON_NETCONF_DIR	$APPDIR/netconf	Dir of netconf plugins

CLICON defines access functions for all system options. Thus, for example, one can access the value of `CLICON_APPDDIR` by `clicon_appdir(h)`.

Appendix C: CLICON CLI callbacks

Name/Example	Description
`cli_run("ntpq -p")`	Execute a shell command given by the argument.
`show_conf_as_text("running ,*$")`	Show the configuration database as CLI commands. The argument determines which database and which keys to print.
`show_conf_as_xml("running ,*$")`	Show the configuration database as XML commands. The argument determines which database and which keys to print.
`cli_merge("system.hostname $hostname")`	Maps cli statements to database entries. When used with tree-reference (@) the argument is autoamtically filled in.
`cli_del("system.hostname $hostname")`	Delete a database entry. When used with tree-reference (@), its argument is filled in autoamtically.
`cli_option_str("CLICON_APPDIR")`	Get the value of a CLICON option.
`cli_quit()`	Exits the CLI
`cli_validate()`	Validate the configuration in the candidate datase
`cli_commit()`	Validates and commits the candidate database into running. The argument defines if a snapshot should be taken.
`delete_all("CANDIDATE_DB")`	Delete a whole database.
`cli_debug((int)1)`	Set debug level.
`compare_dbs()`	Print a diff between candidate and running.
`discard_changes()`	Remove candidate and replace it with running.
`save_config_file("running name")`	Save configuration database to a file. The name of the database is the first word of the parameter and the cli variable name containign the filename is the second.
`load_config_file("filename merge")`	Load configuration database from a file into candidate. The argument contains the cli variable containing the filename and merge or replace operation.

Appendix D: CLICON backend callbacks

A large part of CLICON backend call has to do with getting the values from the database. For this, CLIgen variables are currently used. Many of the following functions are actually CLIgen [1] functions, given here for clarity.

Name/Example	Description
`dbkey2cvec("running_db", "service.ssh")`	Find a key in the db and return its values as a vector of cligen variables. (See CLIgen documentation)
`dbvar2cv("running_db", "service.ssh", "port")`	Get a specific CLIgen variable from specific database key and variable.
`cvec *cvec_new(int len)`	Create and initialize a new cligen variable vector (cvec)
`int cvec_free(cvec *vr)`	Free a cvec
`int cvec_init(cvec *vr, int len)`	Initialize a cligen variable vector (cvec) with 'len' numbers of variables.
`int cvec_reset(cvec *vr)`	Like cvec_free but does not actually free the cvec.
`cg_var *cvec_next(cvec *vr, cg_var *cv0)`	Given an cv in a cligen variable vector (cvec) return the next cv.
`cg_var *cvec_add(cvec *vr, enum cv_type type)`	Append a new cligen variable (cv) to cligen variable vector (cvec) and return it.
`int cvec_del(cvec *vr, cg_var *del)`	Delete a cv variable from a cvec.
`int cvec_len(cvec *vr)`	return length of a cvec.
`cg_var *cvec_i(cvec *vr, int i)`	return i:th element of cligen variable vector cvec.
`cg_var *cvec_each(cvec *vr, cg_var *prev)/*!`	Iterate through all cligen variables in a cvec list
`cg_var *cvec_each1(cvec *vr, cg_var *prev)`	Like cvec_each but skip element 0.
`cvec *cvec_dup(cvec *old)`	Create a new cvec by copying from an original
`int cvec_print(FILE *f, cvec *vr)`	Pretty print cligen variable list to a file
`cg_var *cvec_find(cvec *vr, char *name)`	Return first cv in a cvec matching a name
`cg_var *cvec_find_var(cvec *vr, char *name)`	Like cvec_find, but only search non-keywords
`char *cv_name_get(cg_var *cv)`	Get name of cligen variable cv
`char *cv_name_set(cg_var *cv, char *s0)`	allocate new string from original. Free previous string if existing.
`enum cv_type cv_type_get(cg_var *cv)`	Get cv type
`void *cv_value_get(cg_var *cv)`	Get value of cv without specific type set
`char cv_bool_get(cg_var *cv)`	Get boolean value of cv
`int32_t cv_int_get(cg_var *cv)`	Get integer value of cv
`int64_t cv_long_get(cg_var *cv)`	Get 64-bit integer value of cv
`char *cv_string_get(cg_var *cv)`	Get pointer to cv string.
`char *cv_string_set(cg_var *cv, char *s0)`	allocate new string from original. Free previous string if existing.
`struct in_addr *cv_ipv4addr_get(cg_var *cv)`	Get ipv4addr, pointer returned, can be used to set value.
`uint8_t cv_ipv4masklen_get(cg_var *cv)`	Get ipv4addr length of cv
`struct in6_addr *cv_ipv6addr_get(cg_var *cv)`	Get ipv6addr, pointer returned, can be used to set value.
`uint8_t cv_ipv6masklen_get(cg_var *cv)`	Get ipv6addr length of cv
`char *cv_mac_get(cg_var *cv)`	Returns a pointer to 6-byte mac-address array.
`unsigned char *cv_uuid_get(cg_var *cv)`	Returns a pointer to uuid byte array.
`struct timeval cv_time_get(cg_var *cv)`	Returns a struct timeval by value.
`char *cv_urlproto_get(cg_var *cv)`	Get pointer to URL proto string.
`char *cv_urladdr_get(cg_var *cv)`	Get pointer to URL address string.
`char *cv_urlpath_get(cg_var *cv)`	Get pointer to URL path string.
`char *cv_urluser_get(cg_var *cv)`	Get pointer to URL user string.
`char *cv_urlpasswd_get(cg_var *cv)`	Get pointer to URL passwd string.
`enum cv_type cv_str2type(char *str)`	Translate (parse) a string to a CV type.
`char *cv_type2str(enum cv_type type)/*!`	Translate (print) a cv type to a static string.
`int cv_len(cg_var *cv)`	Return length of cligen variable value (as encoded in binary)
`int cv2str(cg_var *cv, char *str, size_t size)`	Print value of CLIgen variable using printf style formats.
`char *cv2str_dup(cg_var *cv)`	Like cv2str, but allocate a string with right length.
`int cv_print(FILE *f, cg_var *cv)`	Pretty print cligen variable value to a file