Tarasca Project v0.3.2

Documentation

Table of Contents

1. Introduction....2
2. Configuration and Compilation....3
2.1 Download the project....3
2.2 System requirements....3
2.3 Quick configuration and installation....3
2.4 Detailed configuration and compilation....4
2.4.1 Configuration....4
2.4.2 Compilation and execution....5
3. Writing modes and commands....6
3.1 Creating modes and commands....6
3.2 API....10
3.3 Reserved words....12
4. System architecture....13
4.1 The graph....13
5. Trouble-shooting....15
6. Further work (ToDo list)....15

1. Introduction

The Tarasca Project provides a way of implementing user-defined commands in a simple, compact and fast way. It is intended to embedded systems, although it can be used in servers and workstation environments.
It contains the infrastructure for creating a Command Line Interface (CLI) that supports privileged menus, tab completion, command completion, commands history, automatic help generation and some other features.
The developer has to focus only in the functionality of the implemented commands, leaving aside the scanning (lex) and parsing (yacc) details of such commands.
There are many embedded systems that need a special CLI/GUI in which only some type of commands must be executed and special permissions for this commands have to be granted. The capabilities of this system allows to cover these needs and speeding up the time-to-market of the product.

Some of the advantages of the Tarasca Project are:
- Simplicity: Command implementation becomes simple due that no lexical nor syntactical analysis has to be done by the user, thus, there is no need for using lexers (lex) and parsers (yacc).
- Fast development: Command implementation becomes faster due that the user can concentrate only in the function implementation leaving aside the time consuming details of parsers, validations, writing command help, etc.
- Extensibility: The generated executable can be called from a script/program, so it can be used from a GUI. This is still experimental.
- Automatic data type checking: Automatic checking of data type for strings, bools, numbers (integer and decimal), hex, ranges, IP addresses and MAC addresses.
- Friendly CLI behavior: Tab-completion, command-completion, history, handling of CTRL-N keys
- Small foot-print: The final executable has a small foot-print needed for embedded systems with limited resources like flash memory.
- Just one executable: In its default/simplest configuration, only one executable file is needed, no configuration nor log files are used.

The way the system is used can be described in a simple manner (an in-depth explanation is described in section 4):
- Describe the definition of a command with its arguments in a configuration file.
  Eg: A command could be *save-router-conf*.
- Write the function that will actually execute the command in a C file leaving aside the lexical and syntax details.
  Eg: The function that implements the command *save-router-conf* could be *save_conf()*.
- Compile the system.
- Execute it in the target system.
  The *save-router-conf* command will be available in the menu and will execute the defined function *save_conf()* when typed.
- Execute it from a GUI script/program with a system call. The output will be send to *stdout* that can be easily redirected.

The final goal is that every embedded system that needs a CLI and/or a GUI could make use of this system given its useful capabilities that accelerates and simplifies the development process.

The system has been tested in the following platforms:
- i386 with Linux FC3, FC4, FC6 and SuSE 9.1
- MIPS with embedded Linux
- Solaris 8

It should work with other Unix systems and other platforms without major or any modifications.

This project is still in an early stage, thus, A LOT of development has to be done, it has many bugs. Any help in the form of patches, feedback, docs, etc is more than welcome!

2. Configuration and Compilation

2.1 Download the project

The project homepage where it can be downloaded is http://sourceforge.net/projects/tarasca

2.2 System requirements

The requirements for configuring and compiling the system in the host machine are the following (they are normally installed by default in most Linux distributions):

- Perl

Lex (Flex) and Yacc (Bison) could be needed if the files *lex.yy.c, y.tab.c* and *y.tab.h* are removed entering

```
$ gmake clean_lex_yacc
```

These files are already present in the vanilla package.

2.3 Quick configuration and installation

```
$ tar xvfz tarasca-0.3.2.tar.gz
$ cd  tarasca-0.3.2

$ ./configure [OPTION]... [VAR=VALUE]...
$ gmake
```

The executable, called *crish*, is inside the subdirectory /*cli* of the project.

```
$ cli/crish
```

IMPORTANT: If the system was configured with the *--enable-auth* option,
the default password is ***admin123***

2.4 Detailed configuration and compilation

The first step is to decompress the source code and change to the newly created directory:

```
$ tar xvfz tarasca-0.3.2.tgz
$ cd  tarasca-0.3.2
```

2.4.1 Configuration

The *configure* script is used for setting the important variables and options.
This script is not like the 'normal' *configure* provided by *autotools*. This is because the *autotools* are not very friendly when cross-compiling, therefore the *configure* provided is a simple Perl script.

The available variables and options are displayed typing

```
$ ./configure --help
```

The explanation of each variable and option is shown in the following table.

Variable/Option	Description
Installation directories	
--confdir=CONFDIR	Configuration directory where the *tarasca.conf* file resides. This file is used only when the *-enable- auth* and/or *-ena ble-last-login* options are enabled. It contains three parameters: the login-password, the enable-password and the last login date. Default: [./etc]
--commandsdir=CMDDIR	The configuration files and source code of the user-defined functions reside in this directory. These files are needed for generating the final executable file. Once this file is created, these file are not needed anymore. Default: [./commands]
--logdir=LOGDIR	Logs data directory. This file is used only when the *--enable-logs* option is enabled. Default: [./log]

Variable/Option	Description
Compiler, parser and scanner	
--cc=COMPILER	Use compiler COMPILER. Default: [gcc]
--lex=LEX	Use lexical parser LEX. Default: [lex]
--yacc=YACC	Use syntactical scanner YACC. Default: [yacc]
Optional Features	
--enable-auth	Ask for a passwd when executed from a terminal. If this option is enabled, the system will ask for a password right after start-up and after the enable commands. The default password in both cases is **admin123**. Default: [no]
--disable-shell-msg	Display a welcome msg when executed from a terminal. Default: [yes]
--enable-last-login	Display at login time, the date from the last login. Default: [no]
--enable-end-msg	Display a good-bye msg when quitting. Default: [no]
--enable-gui	Enable the capability of being called from another program/script/shell. Default: [no]
--enable-logs	Write logs to a file located at LOGDIR. See option --logdir Default: [no]
--enable-debug	Output generic debug info. Default: [no]
--enable-debug-rhal	Output debug info for the RHAL (project internals). Default: [no]
--enable-debug-shell	Output debug info for the shell (project internals). Default: [no]

Table 2.1 Variables and options of the *configure* script

If the system was configured with the default options (everything disabled) or the directory paths were absolute, *crish* can be copied to any directory in the target system and executed.

The configuration options --confdir, --commandsdir and --logdir can be located in any directory where you have write access. This way --confdir and --logdir can be in different filesystem. For example, the configuration file specified with the --confdir option can be located in a JFFS2, whilst the logs file specified with the --logdir can be located in a tmpfs that would be lost after a system reboot.
The --commandsdir is only used for generating the final exec, it isn't used at execution time.

If the *--enable-gui* option was enabled, the executable, called *crish*, can be called from a script/program/shell. This is described in more detail in section 2.4.2.

2.4.2 Compilation and execution

Compile the source code

```
$ gmake
```

The generated executable, called *crish*, is inside the subdirectory /*cli* of the project and can be executed as follows

```
$ cli/crish
```

IMPORTANT: If *-enable-auth* was set, the default passwords for login and enable are ***admin123***
They can be changed with the *passwd-login* and *passwd-enable* commands, respectively. This commands are provided as part of the default configuration files in the distribution.

If the *-enable-gui* option was enabled, the executable, called *crish*, can be called from a script, shell or program like a Perl/PHP script or a C/C++/Java program with a system() call or equivalent. In this way, a GUI can make use of this system without rewriting the same functionality.

For example, after implementing the command *my_cmd1* in the Tarasca project, it will be available from the CLI in a terminal when calling just *crish*

```
$ cli/crish
tarasca> my_cmd1 item1
Sample output of my_cmd1 with argument item1
tarasca>
```

The same commands *my_cmd1* will be available at the same time to some other script/program/shell, like a web-based GUI written in Perl/PHP, when calling *crish* with the command and arguments as its parameters. The result of the executed command will be sent to *stdout*, so the script/program/shell needs only to redirect the *crish*'s *stdout* to a file/hash/array.

Inside a Perl script:

```
`cli/crish my_cmd1 item1 > output_file`;
```

The file *output_file* will contain "`Sample output of my_cmd1 with argument item1`"

Warning: This is still experimental and it has not been widely tested.

3. Writing modes and commands

One of the main purposes of the Tarasca Project is to simplify and speed up the implementation of commands and the modes to which they belong.
It is strongly suggested to have a look at the samples in the *commands* directory.

3.1 Creating modes and commands

There are four clear steps for writing new modes and commands that are described in detail.

1. Writing modes and commands in the configuration files.

The configuration files are used for declaring modes, commands and arguments. These files must reside inside the *commands* directory because the CLI reads them from this directory when it is executed. The name of the directory can be changed at configuration time with the option –*commandsdir*. By convention, these files have the extension .cli
A good starting point for learning how to write modes and commands is to look at the samples that come inside the *commands* directory.
The structure and priority of a configuration file called *sample.cli* is the following:

```
mode my_mode1 {
       prompt “%h#”;

       command my_cmd1 {
              func = my_func1;
              desc = “Describe what the command does”;

              arg item1 {
                     value = string;
                     prio = mandatory;
                     desc = "Mandatory argument of my_cmd1";
              }

              arg item2 {
                     name = “item”;
                     value = range;
                     prio = optional;
                     desc = “Optional argument of my_cmd1”;
              }
       }
}

mode my_mode2 {
       prompt “%h(my_mode2)#”;

       command my_cmd2 {
              func = my_func2;
              avail = my_mode1;
              desc = “Describe what the command does”;

              arg item1 {
                     name = “item_1”;
                     value = number;
                     prio = optional;
                     desc = "Optional argument of my_cmd2";
              }

              arg item2 {
                     name = “item_2”
                     value = bool;
                     prio = optional;
                      parent = item1;
                     desc = “Optional argument of my_cmd2, depends from arg item1”;
              }
       }
}
```

List 4.1. Configuration file ‘sample.cli’

As can be seen, there are several parts that compose a configuration file and they are explained in the following paragraphs.

Modes

The modes are used for classifying in 'levels' the different types of commands. For example, the commands that configure the system's interfaces could be inside the mode 'iface', whilst the commands that display statistics could be inside the mode 'stats'.
Between one mode and another a password could be required. See the 'enable' command in the samples.

Several modes can be defined inside a configuration file. And the same mode can be defined in several configuration files. In the example above two modes are defined in the same configuration file, but new commands can be added to the same modes in other configuration files.
There can be two types of elements inside a mode: 'prompt' and 'command'.
- prompt: The string defined in the 'prompt' element will be printed each time the shell is ready for receiving commands. In the case that the mode is defined more than once and in each definition has a different prompt, the last one that is read will be used. In case no prompt is defined, a default will be used. For displaying the host name as the prompt use the string *%h*.
- command: Defined in the following paragraph.

Commands

The commands describe the structure of an instruction that will be executed from the CLI/GUI.
Two 'commands' of the same name can exists in two different modes. Otherwise the last one that is read will be used.
A 'command' definition contains four elements: 'func', 'avail', 'desc' and 'arg'.
-func: The name of function that will be called when this command is given.
- avail: Indicates that this command will be available to some other mode that has been previously defined.
- desc: Describes the command. This text could be displayed when asking for help regarding this command.
- arg: Defined in the following paragraph.

Arguments

The argument is a parameter that is given to a command.
There are two types of arguments: 'mandatory' and 'optional'.
The mandatory arguments have to be written always at the beginning of the command definition, then the optional arguments can follow.
The elements of a mandatory argument are the following:
- value: Indicates the type of argument. At the moment the supported types are: strings, bools, numbers (integer and decimal), hex, ranges, IP addresses and MAC addresses.
- prio: Indicated the argument's priority. For this type of argument it must be 'mandatory'.
- desc: Describes of the argument. This text could be displayed when asking for help regarding this argument.
Examples of mandatory arguments are inside 'my_cmd1'.
The elements of an optional argument are the following:
- name: Indicates the string that has to be typed for selecting this argument. In the CLI, after this 'name', the value has to be entered. For example, for executing the command 'my_cmd2' of mode 'my_mode2' with both optional arguments, the following is entered:

```
> my_cmd2  item_1  500  item_2  enable
```

The value of the optional parameter *item_1* is 500 because is declared as number.
The value of the optional parameter *item_2* is *enable* because is declared as bool. The type 'bool' can be written in the form 1/0 or enable/disable.
- value: Indicates the type of argument. At the moment the supported types are: strings, bools, numbers (integer and decimal), hex, ranges, IP addresses and MAC addresses.
- prio: Indicated the argument's priority. For this type of argument it must be *optional*.
- desc: Describes of the argument. This text could be displayed when asking for help regarding this argument.
- parent: Indicates that this optional argument depends on the given one that was declared first. For example, the argument 'item2' depends from 'item1', this means that 'item2' cannot be accepted if 'item1' was not written first. An instruction like the following is not valid:

```
> my_cmd2  item_2  enable
```

whilst this one is valid:

```
> my_cmd3  item_1  7000  item_2  disable
```

2. Including the configuration file
   Once the configuration file has been created, its name is added in the file *tarasca_command_files.conf*. The order in which the files appear is not important anymore. Previous versions needed a specific order.
   For example, the sample file *commands/tarasca_command_file.conf* that comes with the distribution contains three lines that corresponds to one configuration file.

```
user_mode.cli            # Functions available to all modes
privileged_mode.cli      # Privileged functions
config_mode.cli          # System functions
```

   It's important to mention that in the default/basic configuration these files are only needed for generating the final executable *crish*. In this way there is no need for having these configuration files in the production (target) system.

3. Writing the functions
   This is the user-defined function that actually does the work. The functions have to be defined in a *.c* file inside the *commands* directory and their declaration in a *.h* file in the same directory.
   There is no need to include the files in the Makefile since they are included automatically.
   The function must receive as an input parameter the list of arguments given to the command in the CLI or GUI and do not return any parameter.
   For example, a *my_file.c* sample is the following:

```
/*
 * Needed for the printing functions, the graph and generic defines
 * inside the Tarasca project
 */
#include "tarasca.h"

// Declare the global pointer to the list of modes, commands and arguments
extern CLIMatrix  matrix;


void my_func1(SList arg_list) {
    ... code ...
}
```

The *my_file.h* file would be the following:

```
#include "tarasca.h"

void my_func1(SList);
```

4. Generate the new executable
   Compile the whole project executing *gmake* in the root directory of the project distribution.

If the are no errors, the new shell should be in the directory *cli* and is called *crish*.
Execute and enjoy it!

3.2 API

The Tarasca Project provides an API that
- Simplifies an speeds up the acquisition of the commands and their arguments.
- Simplifies the display of data using simple tables and printing functions
- Simplifies the data type checking

There is still A LOT of work to be done in the API, specially the printing functions.

The following paragraphs show the API functions available to the user organized by topic. These functions can be accessed by including the file *tarasca.h* in the user source file.

CLI/GUI arguments acquisition

*int trs_get_param_num (SList, const char *)*
Search in the arguments list the value of the given string. This function must be called when the argument's type is BOOL or NUMBER (int).
If the param's type is BOOL but the parameter is invalid or not found returns -1.
If the param's type is NUMBER but the parameter is invalid or not found returns (1 << 30) * (-1).

*char * trs_get_param_str (SList, const char *)*
Search in the arguments list the value of the given string. This function must be called when the argument's type is STRING, HEX, IPADDR, MACADDR or RANGE. Returns a char * or NULL if

there was an error or was not found.

*double trs_get_param_range (SList, const char *)*
Search in the arguments list the value of the given string. This function must be called when the argument's type is RANGE. If the parameter is invalid or not found returns (1 << 30) * (-1).

*char * trs_get_param_name (SList, const char *)*
Search in the arguments list the value of the given string and return it. Returns a char * if the name was found or NULL otherwise.

*int trs_exists_param (SList, const char *)*
Search in the arguments list the given string. Returns 1 if the string was found, 0 otherwise.

Printing functions

*void trs_print_line (char *)*
Print a single text line adding a '\n' at the end

*void trs_print_str (char *)*
Print a string

void trs_print_nl (int)
Print one or more '\n' depending on the given value.

void trs_print_table_config (short int, ...)
Configure the output table before printing.
The supported types are string '*s*', integer '*d*', unsigned int '*u*', hex '*x*', double '*f*' (float) and long double '*F*' (long float).
For example,

```
print_table_config(2, 's', 30, 'd', 10);
```

Column 1 is of type string (s) and takes 30 spaces to be displayed
Column 2 is of type integer (d) and takes 10 spaces to be displayed
When using this function, the global pointer *ptable* must be declared:

```
extern PrintTable    ptable;
```

void trs_print_row (PrintTable table, ...)
Prints a table row with the format previously given by print_table_config().
If other parameters are given, undefined behavior can occur.
For example,

```
print_table_config(2, 's', 30, 'd', 10);
print_row(ptable, "Value name", value);
```

Mode functions

*int trs_graph_mode_change(char *)*
Change the mode to the given one. Returns 1 if the mode change was successful, 0 otherwise.

Generic configuration file

*char * trs_config_file_param_read(char *filename, char *key)*
Read the value of the given *key* inside the given *filename*.
Return the value of the key as a string.
The format of the filename must be of the type:
graph_mv_set_mode name = value
There could be spaces and tabs before and after each name or value.
Comments start with '#' (hash).
An example of the syntax id the *tarasca.conf* file in the *etc* directory of the project distribution.

*int trs_config_file_param_write(char *filename, char *key, char *value)*
Write *value* of the given *key* inside the given *filename*.
The format of the filename must be of the type:
name = value
There could be spaces and tabs before and after each name or value.
Comments start with '#' (hash).
An example of the syntax id the *tarasca.conf* file in the *etc* directory of the project distribution.

MD5 hash functions

These functions are available only if the option – *enable-auth* was set at configure time.
Examples on how to use these functions are found in *commands/privileged_mode.c*

*void md5_starts(md5_context *ctx)*
Initializes and MD5 context.

*void md5_update(md5_context *ctx, uint8 *input, uint32 length)*
Updates MD5 hash from *input* of length *length* using the context *ctx*.

*void md5_finish(md5_context *ctx, uint8 digest[16])*
Creates and returns the digest in *digest*.

3.3 Reserved words

The following table shows the keywords that are used by the scanner and parser, thus, they can only be used as indicated in section 3.1. User-defined functions cannot have these names.

mode	prio	ipaddr
prompt	parent	networkmask
command	desc	macaddr
avail	bool	range
name	string	optional
value	number	mandatory
func	hex	

Table 3.1. Keyword used by the scanner/parser

4. System architecture

This section is still under development.

4.1 The graph

The main data structure is a graph that contains 3 single-linked lists.

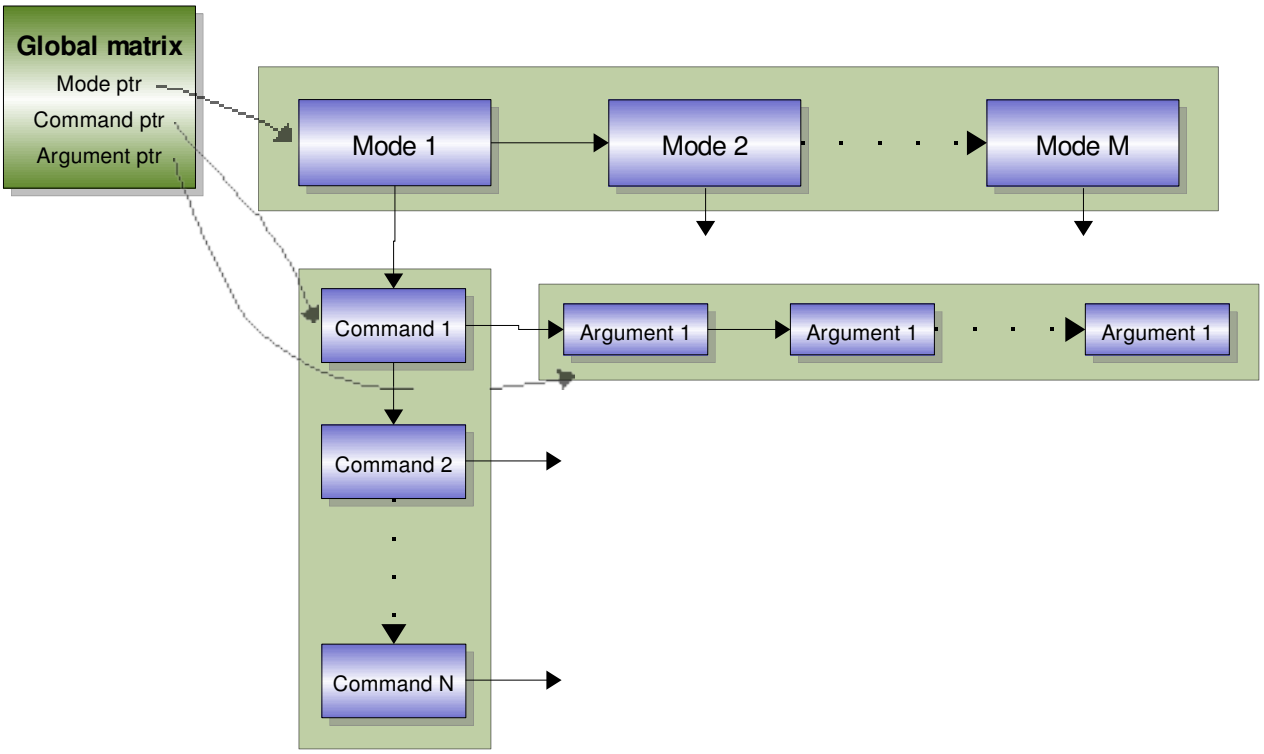

Diagram 4.1. The graph composed of the three different types of lists with the global pointers

The system can have several modes, according to functional levels, some of them may need authorization.

List 1: The first list is composed of modes, each node in the list represent a mode.
Eg. the first node (level) could be for basic commands, second node for privileged (root) commands, third mode for interfaces configuration, and so on.

List 2: Since each mode can have several commands, each node of the mode list has a list of commands. Each node in this list represents a command.
Eg. For the third mode of the previous example, a typical commands would be 'ifconfig' and 'route'.

List 3: Since each command can have several parameters (also called arguments), each node of the command list has a list of arguments. Each node in this list represents an argument.
Eg. The command 'ifconfig' needs as arguments the IP address and netmask.

In order to build the graph and move along it efficiently, there is a global struct called matrix that contains pointers to each one of the three lists. These pointers will move to the mode, command and argument that are been entered at any given moment.

The commands and arguments entered in the terminal are compared against the commands and arguments in this graph. If a match is found, the user-defined function associated to the command is executed.

The program in charge of creating the graph for the first time is *graph_static_gen*.
This program reads the user-defined *.cli* configuration files found in the *commands* directory and creates a 'dynamic' graph. Each node is created according to the obtained data that can be changed without recompiling the source code.
This program is intended to be executed only by the Makefile when building the whole project.

Once the graph is created, a C source code file is written with all the graph information for creating it later from this C source code instead of the *.cli* files.
The purpose of this is that the generated C source code file is compiled and linked with the final executable *crish*. Therefore, the executable does not need any commands configuration file at all.

The program that generates the C file is *graph_static_generator.c*.
The generated C file is called *graph_static.c*. This is the file that will be part of the final executable.
It's is called 'static' because it's nodes are created from the C source code, it's nodes cannot be changed.

This approach provides several advantages such as improved security because the are no configuration files that any user could read. Other advantages are specific for embedded systems: it reduces the system foot-print given that there is no need of flash memory for any configuration file; it saves several read operations from flash memory.

NOTE: If the system is configured with some of the *configure* script options it may need to write to one or more files. These files are not related at all with the commands configuration files.

5. Trouble-shooting

Since this project is still in an early stage there could be beautiful crashes or some strange behavior.
The following list will be an attempt to solve the possible problems that could appear and in the future could be part of the FAQ:

5.1 Command does not appear in shell
A. Check that its name is correct in the configuration files or that the file that defines it is being compiled.

5.2 How can I interact directly with the system's graph?
A. Have a look at the functions declared in *rhal/graph.c*

5.3 How can I display a table?
A. Use the printing functions provided by the API. See section 3.2 and the source code in *rhal/print.c*

5.4 When compiling I get this error:
"g raph_static_gen: FATAL: Could not create dynamic graph
gmake[1]: *** [graph_static_generator.o] Error 255"
A. Probably the program that generates the graph, *graph_static_gen* (see section 4), does not find the *commands* directory. Check that the directory that you gave to the *--commandsdir* option (default is ./commands) at configuration time exists and have the configuration files along with the source code that you wrote.

5.5 When executing *crish* I get an error like this:
f open(): Could not open file... " or "fopen(): No such file or directory"
A. Check that the directory that you gave to *--confdir* or *--logsdir* exists and have the needed files.
This problem could appear when the *--enable-auth*, *--enable-last-login* or *--enable-logs* options are enabled at configuration time.

5.6 The compiler gives an error when using paths with spaces:
The configure script, that edits the Makefiles and the *rhal/config.h* file, does not support spaces yet.

6. Further work (ToDo list)

There is A LOT of work to, the following list is just some of the must important things to do.

- Extensive testing.
- Improve printing functions.
- The configure script doesn't support paths with spaces
- Improve the way the arguments of a command works. Currently, the mandatory arguments must appear before the optional arguments and the optional arguments are not very flexible.
- Fix A LOT of bugs.
- Extend documentation.
- Each mode must have its own command history.
- Tab completion when the first N chars are spaces.

Pedro Aguilar
<paguilar@junkerhq.net>