Linux like Command Line Interface for C/C++ projects

Author

Abhishek Sagar, Juniper Networks

2018

Contents

[Introduction 1](#_Toc512288665)

[Requirements 2](#_Toc512288666)

[Header file Inclusions 4](#_Toc512288667)

[Compilation 6](#_Toc512288668)

[Sample commands 8](#_Toc512288669)

[Terminology 8](#_Toc512288670)

[Command Implementation 9](#_Toc512288671)

[Parsing of LEAF values 11](#_Toc512288672)

[Config Commands Implementation 12](#_Toc512288673)

[Conclusion 12](#_Toc512288674)

# Introduction

This is a Quick guide regarding how you can give your C/C++ linux project a linux like command like interface (CLI). Using this CLI you can interact with the project easily and also automate the project configuration.

Students, especially beginners, when do several linux projects, I find that the first basic difficulty they face is how to interact with the project – especially linux projects which usually do not have GUIs. I also see, students find an alternative and try to build Menu based approach instead as a means to provide front end interface to their linux projects. Well, that works for small projects with limited functionality, but soon fails when projects scales in size. Keeping that in mind, I have made a light-weighted useful C library which you can easily integrate with your project and develop your command project specific commands to interact/control your project/application.

Through this Document, I will tour you quickly using example how you can integrate this library in your project and develop useful commands.

This document do not discuss the internal details regarding design and implementation of the CLI interface. This document serves the purpose to help you to only “USE” this useful library to enable your project to have Command line interface.

# Requirements

First thing to do is to download the library on your linux machine. I assume you have git installed in your machine. If not, download the project from here: <https://github.com/sachinites/CommandParser>. But it is very bad that you do not have Github account.

Those who have git installed can download the project simply using git command:

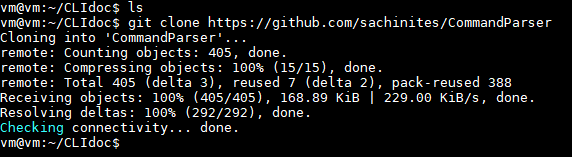


Figure download project using git

The directory *CommandParser* will be created in your current directory. The src code of the CLI project is located in this directory. Also there is a sample file *testapp.c* which is the sample application making use of CLI library. You can also refer to this file regarding how to use the library.

Next, is to build the project.

Go inside the *CommandParser* directory and run ***make***. This will compile the library. It will create an executable ***exe*** and static library ***libcli.a.*** You can run ***exe*** to take a look and feel about the CLI it renders. We will link our application with ***libcli.a*** library to make use of this library.

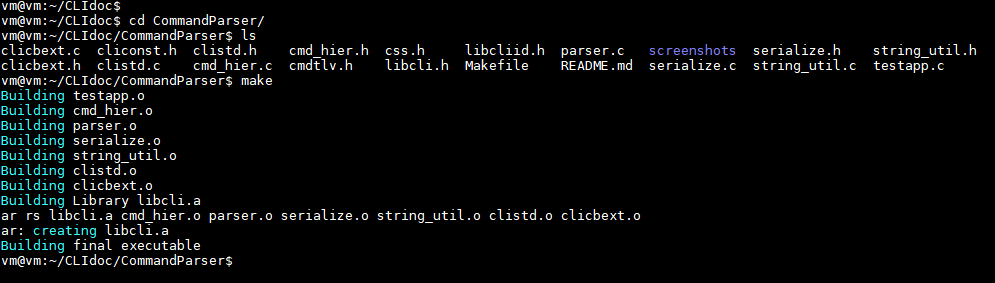


Figure output of make command

There are three files in CommandParser dir namely : main1.c main2.c and main3.c. Cut-paste them outside the CommandParser dir but inside the parent directory as show in figure 3.

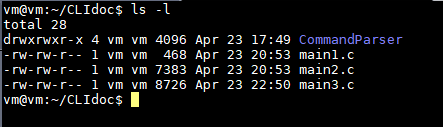


Figure File organization

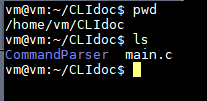


Figure main1.c (correction it is main1.c)

# Header file Inclusions

Libcli library provides two header files as in interface to external projects. In file ***main1.c,*** include the header files as below:

#include "cmdtlv.h"

#include "libcli.h"

In main(), call the below function right at the beginning of main:

init\_libcli(); >> This initialize the library to be used by your application

and, at the end of main() , call the below two APIs :

support\_cmd\_negation(config); >> This adds ‘no’ support to config commands. More on this later.

start\_shell(); >> This finally starts the shell interface. Any code after this function call is dead code.

After ***init\_libcli();*** in main() call the below APIs to import hooks from library.

param\_t \*show = libcli\_get\_show\_hook();

param\_t \*debug = libcli\_get\_debug\_hook();

param\_t \*config = libcli\_get\_config\_hook();

param\_t \*clear = libcli\_get\_clear\_hook();

param\_t \*run = libcli\_get\_run\_hook();

So, at this point, your ***main1.c*** should look like in figure 5 :

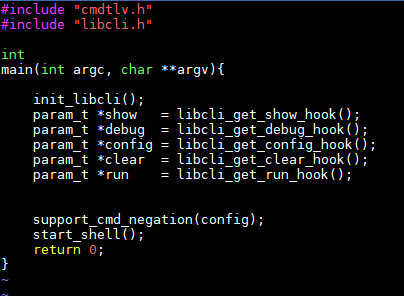


Figure main1.c

Here, ***“show”, “debug”, “config”, “clear”*** and “***run”*** are called hooks. Every project specific command that we will going to implement would start from one of these hooks. Infact, each of these hooks are root of command tree. There will be separate command tree for each hook as a root.

# Compilation

To compile the main1.c , use the following command shown in figure 6.

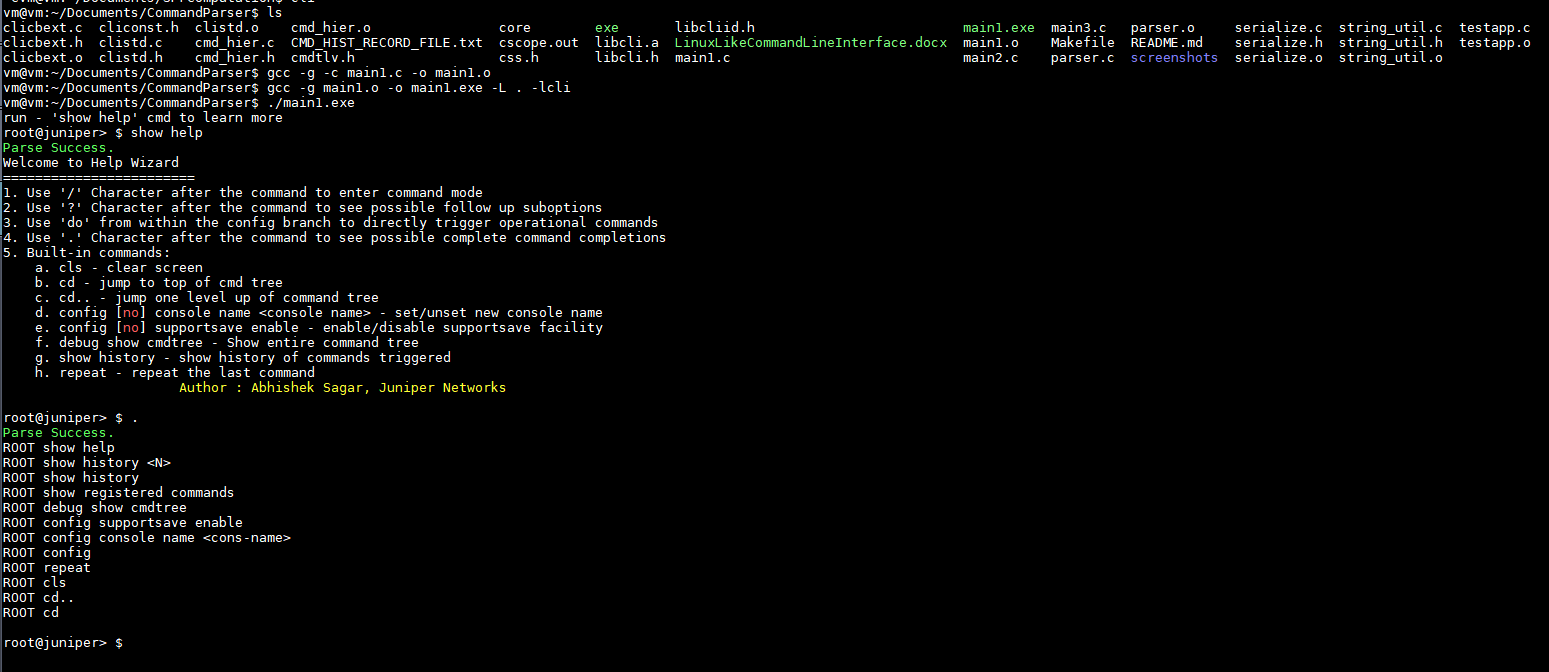


Figure Compilation and run main1.exe

The first command compiles the ***main1.c*** to create object file ***main1.o***. ***–I*** option tells compiler where to find header files included in ***main1.c.***

The second command links our project with libcli library and create final executable – ***main1.exe*.** You can run this executable and play around the same default commands implemented. Also, use ***show help*** to get yourself familiar with the CLI. Try out “?”, “.”, “/” features. See the figure 6.

# Sample commands

To provide as an example of how to implement the commands, we will take following two commands for illustration:

***CMD1 : show node <node\_name::string>***

***CMD2 : config node <node\_name::string> loopback <lo ipaddress>***

# Terminology

Get familiar with the terminology we will be using in the rest of the project.

CMD1 is composed of three words : “show”, “node” and “<node\_name>”. These words are called ***params.*** “show” is a root param (hook) and is provided by library through API ***libcli\_get\_show\_hook();*** The *params* are divided into two types : ***CMD params*** and ***LEAF params*.**

***CMD params*** : CMD params are those which are mere keywords in the command. “node” in CMD1 and CMD2 is CMD param. “loopback” is a CMD param in CMD2.

**LEAF params** : LEAF params are those which can have values. <node\_name> in CMD1 and CMD2 are leaf params. <lo ipaddress> is a leaf param in CMD2.

***Example of triggering CMD1 :***

show node R1

show node R2

show node R3

***Example of triggering CMD2 :***

show node R1 loopback 122.1.1.1

show node R2 loopback 122.2.1.1

For every Command you type, and press enter, there should be some functionality to be invoked. These functionalities are nothing but functions of our external projects. We will register our project specific function/routines to the commands so that, these commands, when triggered by the user result in executing our application routines/functions.

# Command Implementation

Now, without wasting any further time, let us straight away implement our CMD1.

Before that, first we need to discuss the param constructor which actually initialize the new param – CMD or LEAF.

The constructor function is ***init\_param.*** The arguments to this init\_param is explained below. Each argument to the constructor is passed in the context whether the param is CMD or LEAF param.

*void*

*init\_param(param\_t \*param, /\* pointer to param\_t variable\*/*

*param\_type\_t param\_type, /\* CMD if it is a CMD param , LEAF if it is a leaf param\*/*

*char \*cmd\_name, /\* <command name> for cmd param | always NULL for LEAF param\*/*

*cmd\_callback callback, /\* Callback field, applicable to both CMD and LEAF param. Can be NULL\*/*

*user\_validation\_callback user\_validation\_cb\_fn, /\* always NULL for CMD param | Can be NULL Or valid function pointer for LEAF param \*/*

*leaf\_type\_t leaf\_type, /\* always INVALID for CMD param | leaf type (discuss shortly)\*/*

*char \*leaf\_id, /\* always NULL for CMD param , some <STRING> for LEAF param\*/*

*char \*help) ; /\* Help String \*/*

In the above definition of init\_param() API, I have explained the meaning of each argument. Though some arguments needs detailed discussion.

*Arg : cmd\_callback callback;*

This field is a pointer to applicable specific function which you want to invoke when user type out the command on shell upto this param and hit enter. If this field is left blank, the command exactly upto this param is said to be incomplete command.

The function prototype MUST be:

int <function\_name> (param\_t \*param, ser\_buff\_t \*tlv\_buf, op\_mode enable\_or\_disable) ;

where, ***ser\_buff\_t*** and ***op\_mode*** are data types provided by library. We will discuss their use later. This function must always return 0.

*Arg : user\_validation\_callback user\_validation\_cb\_fn;*

This field is a pointer to application specific function which you want to invoke in order to validate the value passed by the user for this param. This field is meaningful only for LEAF param.

The function prototype MUST be :

int <function\_name> (char \*value);

The value of the leaf param as typed out by the user on shell is passed as argument to this function as string. You can write your own logic how to validate this value further. This function must return either VALIDATION\_SUCCESS or VALIDATION\_FAILED as return value.

Other fields in the init\_param() have been explained already.

Now, having roughly gone to the above constructor. I would recommend you to take a look at ***main2.c*** file in which I have implemented both CMD1 and CMD2 with good enough comments. Just compile main2.c like you compiled main1.c and run the executables. Trigger the new commands CMD1 and CMD2 and experience the library. Just trigger ***“.”***  and it will show all complete commands.

# Parsing of LEAF values

Now, you must have seen function *node\_loopback\_callback\_handler()* was invoked when you ran CMD2. CMD2 has two Leafs in it : node\_name and loopback\_address. We should be able to fetch the values of these two leaves in application code along with command code. For this purpose, we have macros.

For extracting command code , use macro : EXTRACT\_CMD\_CODE

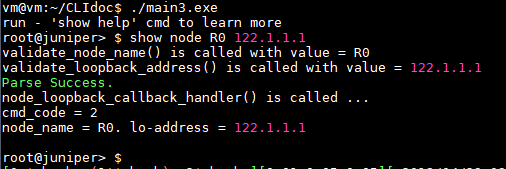
For Iterating over leaf values:

TLV\_LOOP\_BEGIN

TLV\_LOOP\_END

We implement this functionality on top of file main2.c in file ***main3.c***. Take a look at the function: ***node\_loopback\_callback\_handler()*** implementation in main3.c.

When we trigger CMD2, we get the following output as shown in figure 7:



Figure

You can see, command code recvd is 2 which is CMDODE\_SHOW\_NODE\_LOOPBACK (#define constant), and both the leaf values are printed. Please go through file ***main3.c*** and see how  ***node\_loopback\_callback\_handler()*** function is implemented.

Different leaf type supported are defined in CommandParser/libcliid.h file. Have a look.

# Config Commands Implementation

We come to the last section of this document: Implementing the config commands. Implementing config commands is no different than show commands, except the little catch.

When user configure something in application through config commands, at some point of time or later, he would like to un-configure the same configuration also. Config commands are intended to update/modify the application state, unlike show commands.

There is already one inbuilt config command: **config console name <name>**

Try out this command and notice the change in cli prompt. Now un-configure the same using negate command:

**config no console name <name you configure>.** Notice the change again in cli prompt.

Thus, What user configure also needs to be un-configured at some point of time or whenever user wishes. All config commands can be negated using **no** keyword after *config* key word.

For your college/personal projects, you may want to change the name of the prompt as per your requirement. For this, modify the file ***CommandParser/cliconst.h*** and change the following line. You will have to recompile the library after this change to take effect.

*#define DEFAULT\_DEVICE\_NAME "root@juniper".*

Lastly, how application code would know whether the config command triggered tis o configure Or unconfigure something. For this purpose, the last parameter of application routine is used. See the function: *node\_loopback\_callback\_handler()* in which the last parameter is **op\_mode enable\_or\_disable.**

The value of **enable\_or\_disable** is set to **CONFIG\_ENABLE** if positive configuration command is triggered**,** else set to **CONFIG\_DISABLE** for negative configurationcommand. If the command is *show/clear/debug/run* then this parameter is of no use and set to **OPERATIONAL.** Note that for config and its complementary negate command, you receive the same set of leaf values and command code.

Try implement one config command, and run negative version of it as an exercise.

# Conclusion

This library will give you quick Command like interface to enable you to provide CLI interface to your any C/C++ projects. Very handy and light weighted library. Isn’t it good ??

Try out different easy utilities provided by library. See the output of ***show help***command.

Src code is with you – you are the new owner of this library – modify/extend/replicate as you wish. From now, never dare to show your linux C/C++ projects with kiddo Menu interface, and show the CLI interface with all pride.

**Thanks.**