VITESSE®

ICLI Command Generation Guide

Application Note

AN1047

Revision 2.9

Oct 2014

Confidential

Vitesse
Corporate Headquarters
741 Calle Plano
Camarillo, California 93012
United States

www.vitesse.com

Vitesse Semiconductor Corporation ("Vitesse") retains the right to make changes to its products or specifications to improve performance, reliability or manufacturability. All information in this document, including descriptions of features, functions, performance, technical specifications and availability, is subject to change without notice at any time. While the information furnished herein is held to be accurate and reliable, no responsibility will be assumed by Vitesse for its use. Furthermore, the information contained herein does not convey to the purchaser of microelectronic devices any license under the patent right of any manufacturer.

Vitesse products are not intended for use in life support products where failure of a Vitesse product could reasonably be expected to result in death or personal injury. Anyone using a Vitesse product in such an application without express written consent of an officer of Vitesse does so at their own risk, and agrees to fully indemnify Vitesse for any damages that may result from such use or sale.

Vitesse Semiconductor Corporation is a registered trademark. All other products or service names used in this publication are for identification purposes only, and may be trademarks or registered trademarks of their respective companies. All other trademarks or registered trademarks mentioned herein are the property of their respective holders.

Copyright © 2013 Vitesse Semiconductor Corporation

TERMS OF USE

The information provided by Vitesse Semiconductor Corporation ("Vitesse") in this document pursuant to these terms ("Agreement") is intended for illustrative purposes only. All information provided herein is subject to change at any time without notice. The information provided, including but not limited to Sample Code ("Software"), is protected by United States and other applicable copyright laws and international treaties. Vitesse does not grant You any license, explicitly or implicitly, under any trademark, patent, copyright, mask work protection right, trade secret or any other intellectual property right.

ALL INFORMATION, INCLUDING BUT NOT LIMITED TO THE SAMPLE CODE ("CODE ") SUPPLIED, IS PROVIDED STRICTLY "AS-IS" WITH NO WARRANTIES OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, MADE WITH RESPECT TO THE INFORMATION TO INCLUDE THE CODE AND ALL ACCOMPANYING WRITTEN MATERIALS, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. YOU ASSUME THE ENTIRE RISK AS TO THE QUALITY, ACCURACY, AND PERFORMANCE OF THE INFORMATION, AND YOU ASSUME ANY AND ALL RISK AND LIABILITY FOR ANY ACTIONS TAKEN BY YOU ON THE BASIS OF ITS ANALYSIS OR OTHER USE OF THE INFORMATION, INCLUDING BUT NOT LIMITED TO MODIFICATIONS TO YOUR PRODUCTS IN LIGHT OF SUCH USE OF INFORMATION, AND YOU HEREBY ACKNOWLEDGE THAT VITESSE SHALL HAVE NO RESPONSIBILITY OR LIABILITY AS A RESULT OF YOUR USE OF INFORMATION PROVIDED HEREUNDER.

IN NO EVENT SHALL VITESSE BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF USE OR INABILITY TO USE THE CODE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This Agreement is governed by the laws of the State of California, without regard to principles of conflicts of laws. Each provision of this Agreement is severable. If a provision is found to be unenforceable, this finding does not affect the enforceability of the remaining provisions of this Agreement. This Agreement is binding on successors and assigns. By accessing the information contained in or referenced by this document, You acknowledge that You have read this Agreement, that You understand it, that You agree to be bound by its terms, and that this is the complete and exclusive statement of the Agreement between You and Vitesse regarding the information and Code.

Copyright © 2013 Vitesse Semiconductor Corporation

Contents

1 Introduction

1.1 Overview

In order to use a command on ICLI engine, the command needs to be registered into ICLI engine. This document shows how to write script files to auto-generate C and H files for command auto-registration.

The steps are as the following chapters.

- Generation Flow.
- Writing Script.
- Command Generation.

1.2 Audience

This document is for software and application developers who need to understand and use the ICLI engine.

1.3 Revision History

Revision	Date	Reviewers	Description
02-09	30 Oct 2014	CP	1. Update by 3.60 Mass.
02-08	09 Oct 2014	CP	1. Section 3.1.8, Loop.
02-07	29 Aug 2013	CP	1. Section 3.2.4, add 2 new tags, GOTO_MODE and SUB_MODE. 2. Appendix C, modify '?' usage.
02-06	15 Aug 2013	CP	1. Appendix C, add Ctrl-Q to display full command syntax.
02-05	31 Jul 2013	CP	1. Appendix A, list variables allow descending order.
02-04	30 Jul 2013	CP	1. Appendix A, list variables allow incremental only, not equal and not decremented.
02-03	08 Jul 2013	CP	1. Section 3.2.4, update RUMTIME usage. 2. Appendix A, add new variables, <clock_id>, <hexval>, <vword>, <switch_id>, <switch_list>. 3. Appendix C, add Ctrl-w key and remove ESC key.
02-02	21 Jan 2013	CP	1. Appendix A, add 3 new variable types, <ipv4_abc>, <dword>, <fword>. 2. Appendix A, modify descriptions for <a-b> and <a~b>.
02-01	04 Dec 2012	CP	1. Appendix A, add 1 new variable type, <ipv4_nmcast>. 2. Appendix C, created to list short-cut keys.
02-00	26 Oct 2012	CP	1. Section 3.2.5, describe the syntax of random optional. 2. Section 3.2.6, describe the syntax of random must.

Revision	Date	Reviewers	Description
01-09	24 Oct 2012	CP	1. Section 3.2.4, give more descriptions for RUNTIME tag. 2. Appendix B.1, created for multiple optional begin.
01-08	12 Oct 2012	CP	1. Section 3.1.5, modify repeat syntax for { [a|b|c] … } 2. Appendix A, add 1 new variable type, <hostname>.
01-07	27 Sep 2012	CP	1. Appendix A, add 4 new variable types, <dpl>, <dscp>, <oui> and <pcp>. 2. Appendix B, created for ICLI parsing error reference guide. 3. Section 3.1.6, created to describe the design limitation. 4. Section 3.2.4 Export segment, removed.
01-06	18 Sep 2012	CP	1. Appendix A, add 2 new variable types, <port_type_id> and <port_type_list>.
01-05	28 Aug 2012	CP	1. Appendix A, add 3 new variable types, <ipv4_prefix>, <ipv6_prefix> and <hhmm>.
01-04	21 Aug 2012	CP	1. Section 3.1.5, give the examples for all syntax. 2. Appendix A, modify the descriptions of word, kword, string and line. 3. Appendix A, modify the C types of range_list, a~b, port_list, vlan_list.
01-03	13 Jul 2012	CP	1. Section 3.2.5, add 2 command properties, `ICLI_CMD_PROP_LOOSELY` and `ICLI_CMD_PROP_STRICTLY`.
01-02	18 May 2012	CP	1. Section 3.2.5, multiple command modes in a command. 2. Section 3.2.5 add new tags. • FUNC_NAME • FUNC_REUSE • NO_FORM_DOC_CMD_DESC • NO_FORM_DOC_CMD_DEFAULT • NO_FORM_DOC_CMD_USAGE • NO_FORM_DOC_CMD_EXAMPLE • NO_FORM_VARIABLE_BEGIN • NO_FORM_VARIABLE_END • NO_FORM_CODE_BEGIN • NO_FORM_CODE_END • DEFAULT_FORM_DOC_CMD_DESC • DEFAULT_FORM_DOC_CMD_DEFAULT • DEFAULT_FORM_DOC_CMD_USAGE • DEFAULT_FORM_DOC_CMD_EXAMPLE • DEFAULT_FORM_VARIABLE_BEGIN • DEFAULT_FORM_VARIABLE_END • DEFAULT_FORM_CODE_BEGIN • DEFAULT_FORM_CODE_END

Revision	Date	Reviewers	Description
01-01	04 May 2012	CP	1. New syntax, Repeat "…" in Section 3.2.6. 2. New variable, <line>. 3. New feature, Section 3.1.5 Defining Constant String. 4. Remove MODE_CMD. 5. Remove FUNC_NAME. 6. Remove Section 4 Command Generation Utility. 7. Remove Appendix B.
00-09	09 Mar 2012	CP	2. Revise for training course.
00-08	27 Dec 2011	CP	1. Section 3.1.4: Add random optional.
00-07	14 Oct 2011	CP	1. Add new variable type, <a~b>.
00-06	16 Sep 2011	CP	1. Documentation mechanism.
00-05	28 Jul 2011	CP	1. Appendix A: add <port_type>, modify <port_id> and <port_list>.
00-04	14 Jul 2011	CP	1. Syntax change: change the use of [] and {}, that is, [] for optional and {} for mandatory.
00-03	20 Jun 2011	CP	1. Section 5: add.
00-02	24 May 2011	Rene	1. 151 revisions. Please refer to AN1047-00-02-ICLI _Command_Generation_Guide_RBN20110524 .doc
00-01	18 May 2011	Srinivas	1. Appendix A: add a variable type, double, 64-bit floating point.
00-00	13 May 2011	CP	Initial release.
	13 May 2011	Srinivas	1. Section 3.1, 3.2, 3.2.2, 3.2.3, 3.2.4, 3.2.5: English corrections. 2. Section 3.2.1: change name of Global Segment to Module Segment. 3. Section 3.2.3: add constant and data type declarations in Function Segment. 4. Section 3.2.5: add more descriptions for CODE sub-segment. 5. Chapter 4: the layout of directory is modified. 6. Appendix B: add example for two ACL commands.

2 Generation Flow

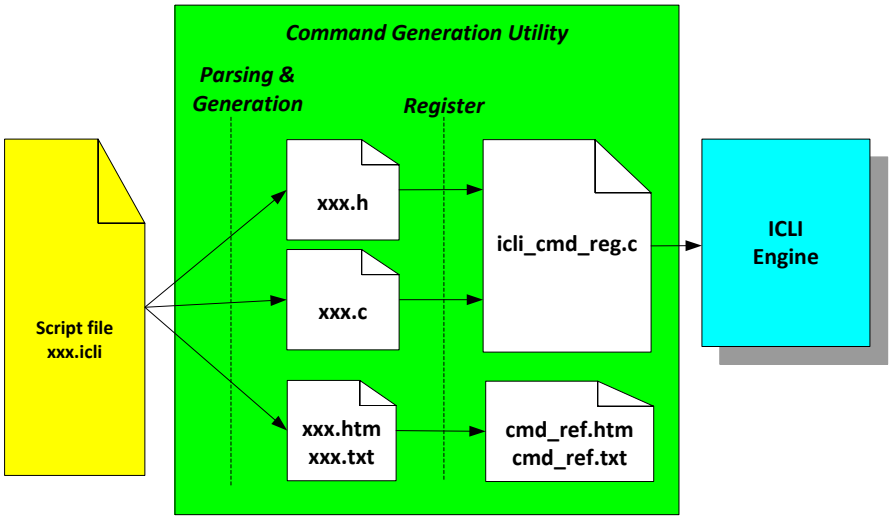

Figure 1. Command Generation Flow.

Figure 1 describes the flow of command generation. The procedures are as follows.

1. Module designer writes the script file (xx.icli).
2. Use command generation utility to
    a. *Automatically* generate the corresponding C and H files (xx_icli.c and xx_icli.h).
    b. *Automatically* register commands into ICLI engine (icli_cmd_reg.c).
    c. *Automatically* generate the command reference guide in HTML (cmd_ref.htm).

By writing script files rather than coding directly in C have some advantages, among others the following:

1. Easy writing.

    The script file makes module designer care about only the implementation of commands. All others of command registrations, parsing, variable types, and memory utilization will be processed by command generation utility.

2. Improve coding effort, reduce coding cost.

    Because the module designer only writes a small portion of source code for the implementation and the other source code is generated by the utility, the coding errors are minimized and as a result, the coding cost is reduced and the efforts are improved.

3. Be able to ignore the implementation details, including coding convention.
4. Generate command reference guide automatically.

3 Writing Scripts

3.1 Command Syntax

Command := keyword word word …

word := keyword or variable

keyword := a constant word

variable := <variable type>

A command consists of several words and the space is used to separate each word. A word can be a keyword or a variable. The first word must be a keyword and can not be a variable.

The keyword is a constant word that user must input literally.

The variable is a variable type enclosed by a set of angle brackets (<>) and the variable type represents the type of the user input.

ICLI engine will parse and check whether or not the user input meets the corresponding variable type.

For example, a command is "`ip address <ipv4_addr> <ipv4_netmask>`". This command has 4 words, "`ip`", "`address`", "`<ipv4_addr>`" and "`<ipv4_netmask>`". "`ip`" and "`address`" are keywords. "`<ipv4_addr>`" and "`<ipv4_netmask>`" are variables. So, the user must input strings to meet the type of `<ipv4_addr>` and `<ipv4_netmask>`. The valid user input could be "*ip address 10.1.1.1 255.255.0.0*". "10.1.1.1" is for `<ipv4_addr>` and"255.255.0.0" is for `<ipv4_netmask>`.

In addition, there is some syntax provided by the ICLI engine. This syntax allows module designer to design his/her commands with flexibility and efficiency.

3.1.1 Variable "< >"

A word enclosed by "< >" means this is a variable, and the user needs to input a string that meets the variable type. Variable types supported by the ICLI engine are described in Appendix A.

3.1.2 Exclusive Or "|"

An exclusive or tells the user that he can choose only one of the words at most. This should be used with Mandatory "{}" or Optional "[]".

3.1.3 Mandatory "{ }"

Exactly one of the words enclosed in braces ({}) must be input. Individual words must be separated by the exclusion operator (|). For example, a command "`a { b | c } d`".

Valid user inputs:

```
a b d

a c d
```

Invalid user input:

```
a d
```

Note that if "{}" does not cooperate with "|" then it is legal but will not have any effect.

3.1.4 Optional "[ ]"

At most one of the words enclosed in square brackets ([]) must be input. Individual words must be separated by the exclusion operator (|). For example, a command "`a [ b | c ] d`".

Valid user inputs:

```
a b d

a c d

a d
```

3.1.5 Random Optional "[] [] … []"

If there are some optionals that are continuous without any mandatory, then it is called random optional and the input can be out of sequence. For example, a command "`a [b] [c] [d]`", then "`[b] [c] [d]`" is random optional.

Valid user inputs:

```
a

a d

a c b

a d b

a b c d

a d c b
```

If you want one or some specific optional is at the fixed location, then {} can be used for that. For example, a command "`a [b] {[c]} [d] [e]`", then "`[b] {[c]} [d] [e]`" is not a random optional because `{[c]}` is required at that location, but "`[d] [e]`" is a random optional.

Valid user inputs:

```
a

a c

a b c

a b e d

a c e d

a b c e d
```

Invalid user inputs:

```
a c b
```

3.1.6 Random Must "{[] [] … []}*n"

The random optional allows no input. For example, a command "`a [b] [c] [d]`", then the user input "`a`" is valid because `b`, `c` and `d` are not necessary. However, sometimes, if you need random optional and also want at least one word inputted, then the syntax of **random must** provides the feature. This syntax is to enclose random option by manadory, `{ random optional }*n`, and to use `*n` to tell at least `n` words in random optional should be inputted, where `*n` must follow `}` and no space is allowed, and `n` is for single digit only. For example, a command "`a {[b] [c] [d]}*1`", then "`{[b] [c] [d]}*1`" is random must that at least one of `b`, `c` and `d` must be inputted.

The followings are valid random must syntax.

```
a {[b] [c] [d]}*1

a {[b] [c] [d]}*3

a {[b] [c] [d]}*9

a {b | {[c] [d]}*1}

a {[b] | {[c] [d]}*1}

a {b | {[c] [d {[e] [f] [g]}*2]}*1}
```

The followings are not valid random must syntax.

```
a {[b] [c] [d]} *1   => the space is not allowed between } and *1

a {[b] [c] [d]}*10   => *10 is more than 1 digit

a {[b] [c] [d]}*1a   => *1a is not single digit
```

Note that although these commands are not valid random must syatax, they still are valid command. But, `*1`, `*10` and `*1a` are regarded as separated keywords.

On the other hand, if `n` is larger than the total number, `m`, of random optional, then `m` will be applied. For example, `a {[b] [c] [d]}*9` is valid and it is the same with `a {[b] [c] [d]}*3` and means all `b`, `c` and `d` should be hiited.

3.1.7 Repeat "…"

The syntax is to repeat times in [|] or {|} automatically. The number of repeat times is the number of seletion in [|] or {|}.

The examples for all syntax are as follows.

{{a|b|c} … } is equivalent to **{a|b|c} {a|b|c} {a|b|c}.**

{[a|b|c] … } is equivalent to **{{a|b|c} [{a|b|c} [{a|b|c}]]}.**

[{a|b|c} …] is equivalent to **[{a|b|c} {a|b|c} {a|b|c}].**

[[a|b|c] …] is equivalent to **[{a|b|c} [{a|b|c} [{a|b|c}]]].**

3.1.8 Loop "( )*N"

The sub command is enclosed by () and N means the sub command can be iterated 1..N times. The corresponding variables will be declared as arrays so you can use array index to get the values. The following code is an example.

```
CMD_BEGIN

COMMAND = debug loop ( uint <uint> )*3

...

CMD_VAR =

CMD_VAR =

CMD_VAR = b

CMD_VAR = value

...

CMD_END
```

Then the variables will be generated as follows.

```
BOOL   b[3];

u32    value[3];
```

Therefore, you can use array index to get the user input for each loop input.

On the other hand, the number of loop, N, can be a constant defined in code as follows.

```
COMMAND = debug loop ( uint <uint> )*_CONST_NUM_
```

Where `_CONST_NUM_` is defined in the ICLI file or other header file.

3.1.9 Limitation

The limitation on ICLI engine parsing is it does not allow the same words that overlap in mandatory{} or optional[].

For example (1), 2 commands,

```
snmp client version <uint>

snmp { server | client } address <ipv4_ucast>
```

Then, the keyword '`client`' is overlapped illegally because '`client`' is in mandatory{} in the second command.

The example (2), 2 commands,

```
snmp client version <uint>

snmp [client] address <ipv4_ucast>
```

Then, the keyword '`client`' is still overlapped illegally because '`client`' is in optional[] in the second command.

The solution is as follows.

Example (1) is re-designed to the following 2 commands.

```
snmp client { version <uint> | address <ipv4_ucast> }

snmp server address <ipv4_ucast>
```

Example (2) is re-designed to the following 2 commands.

```
snmp client { version <uint> | address <ipv4_ucast> }

snmp address <ipv4_ucast>
```

In other words, the concept is to use the same flat prefix as most as possible. "`snmp client`" is the same flat prefix in the examples.

Another example is as following 5 commands.

```
a b x

a c x

a d x

a b y

a c y
```

Because of the limitation, the following design will not be allowed.

```
a { b | c | d } x

a { b | c } y
```

Instead, the following design, from left to right, is allowed.

```
a b { x | y }

a c { x | y }

a d x
```

3.2 Script File

The script filename extension is "icli", for example, acl.icli. The script file is composed of segments. Each segment is composed of tags. If the tag has a value, then you can use '=' to assign the tag value, i.e. "`TAG = TAG_VALUE`". If `TAG_VALUE` is omitted, it corresponds to not specifying TAG at all. If the `TAG_VALUE` is long, you can use '\' to concatenate the next line.

A segment is always enclosed by two tags, `XXX_BEGIN` and `XXX_END`, and may have sub-segments inside.

If you want to comment a line, you can use '**!**' or '**#**' or "**//**" at the beginning of the line.

There are five segments in the script file. They are **Module**, **Include**, **Function** and **Command**. Only the **Command** segment has sub-segments, **VARIABLE** sub-segment and **Code** sub-segment. Figure 2 shows the layout of the script file.

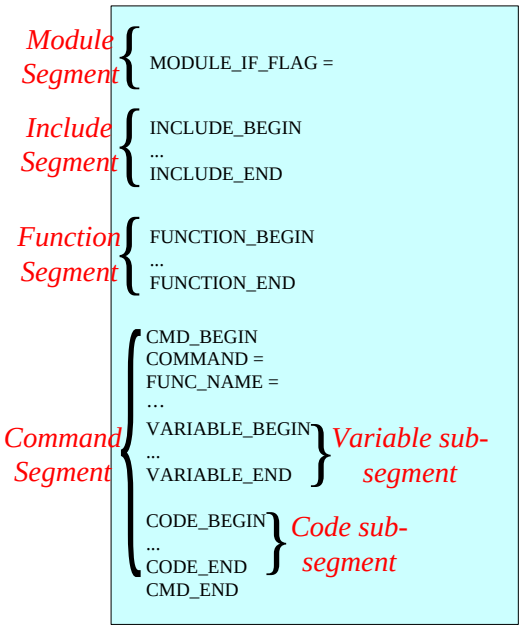

Figure 2. Layout of the script file.

The subsequent sections will explain them one by one along with their supported tags respectively.

3.2.1 Module Segment

This segment controls module-wide settings.

There is only one tag for this segment.

MODULE_IF_FLAG =

(Optional)

This tag value is used to decide whether all commands of this module are registered or not and the generated C/H file is compiled or not.

The tag value is appended to an `#if` directive in icli_cmd_reg.c and the generated C/H files. Therefore, if the tag value does not evaluate to TRUE, the commands will not be registered and even the source code will not be compiled.

Example:

In stp.icli, tag value is VTSS_SW_OPTION_STP.

```
MODULE_IF_FLAG = defined(VTSS_SW_OPTION_STP)
…
…
```

In icli_cmd_reg.c, the auto-registration is enclosed by that tag value.

```
…
#if defined(VTSS_SW_OPTION_STP)
    &stp_icli_cmd_register,
#else
    NULL,
#endif
…
```

In generated C/H files, the whole context is enclosed by that tag value.

```
#if defined(VTSS_SW_OPTION_STP)
…
…
…
#endif
```

3.2.2 Include Segment

In this segment, you specify the include files required to compile the body of the generated C file(s).

"`icli_api.h`" is auto-included by the generator and should not be specified here.

The content will be exactly pasted to the generated C file.

There are two tags for this segment.

INCLUDE_BEGIN

(Optional)

Marks the beginning of **Include** segment.

INCLUDE_END

(Mandatory if **INCLUDE_BEGIN** exists)

Marks the end of **Include** segment.

Example:

```
INCLUDE_BEGIN
#include <stdio.h>
#if STP_MODE_RSTP
#include "rstp.h"
#endif
INCLUDE_END
```

This content will be pasted exactly to the generated C file.

3.2.3 Function Segment

This segment allows the designer to write not only local functions that will be used in the **Command** segment.

In addition, if you need constants, macros, data types, or static variables for the functions or the command bodies, you may declare them in this segment.

The content will be exactly pasted to the generated C file.

There are two tags for this segment.

FUNCTION_BEGIN

(Optional)

Marks the beginning of function segment.

FUNCTION_END

(Mandatory if **FUNCTION_BEGIN** exists)

Marks the end of function segment.

Example:

```
FUNCTION_BEGIN
#define _IS_TRUE(x)    ((x) == TRUE)
static i32    svariable;
static i32 _vid_get(void) {
    return vlan_id;
}
BOOL vid_set(i32 vid) {
    vlan_id = vid;
    return TRUE;
}
FUNCTION_END
```

This content will be pasted exactly to the generated C file.

3.2.4 Command Segment

This segment defines the implementation of the command, one segment for one command. The implementation contains privilege, work mode, command property, help, and execution instance.

The followings are the tags in **Command** segment.

CMD_BEGIN

(Mandatory)

Marks the beginning of command segment.

CMD_END

(Mandatory)

Marks the end of command segment.

DOC_CMD_DESC =

(Optional)

The description about the command describes what the command is for.

It will be displayed in generated HTML file. Please refer to Figure 3.

If the description is too long in one line, you can use '\' to concatenate lines.

If more than one paragraph, you can use multiple **DOC_CMD_DESC**.

DOC_CMD_DEFAULT =

(Optional)

The default value of this command.

It will be displayed in generated HTML file. Please refer to Figure 3.

DOC_CMD_USAGE =

(Optional)

The description about the command describes how to use the command.

It will be displayed in generated HTML file. Please refer to Figure 3.

If the description is too long in one line, you can use '\' to concatenate lines.

If more than one paragraph, you can use multiple **DOC_CMD_USAGE**.

DOC_CMD_EXAMPLE =

(Optional)

The examples how to use the command.

It will be displayed in generated HTML file. Please refer to Figure 3.

If the description is too long in one line, you can use '\' to concatenate lines.

If more than one paragraph, you can use multiple **DOC_CMD_EXAMPLE**.

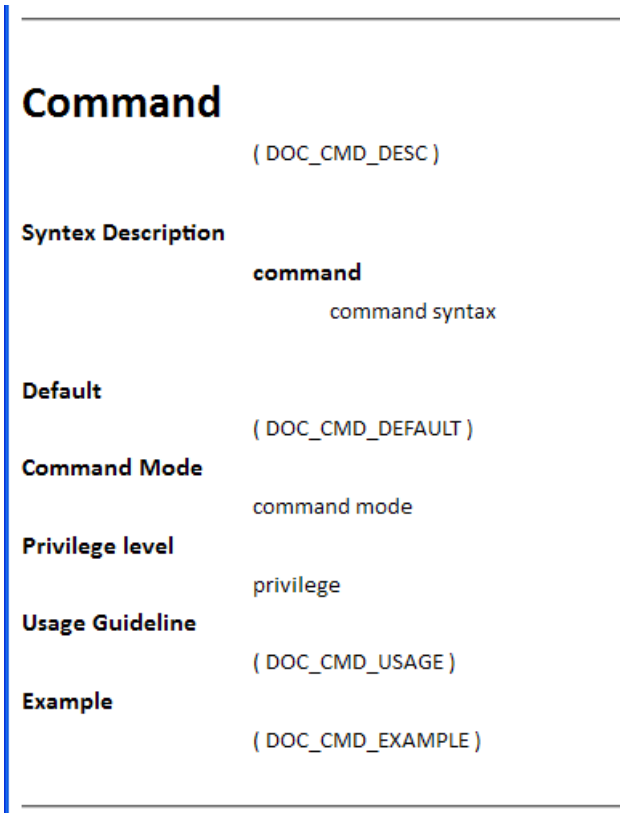

Figure 3. HTML file generated.

COMMAND =

(Mandatory)

Command string with syntax. You can flexibly define the command string with the syntax of mandatory ([]), optional ({}), or (|) and variables (<>).

FUNC_NAME =

(Optional)

Name of command execution function. ICLI command generation utility will use this **FUNC_NAME** to automatically generate a corresponding static function in generated C file. And, this corresponding static function is the command execution function that will be executed when the user inputs a valid command.

If **FUNC_NAME** is not defined, then ICLI command generation utility will randomly generate a name.

FUNC_REUSE =

(Optional)

Name of reused execution function of another command. The use of **FUNC_REUSE** is to reuse the code body of another command, where code body includes Variable sub-segment and Code sub-segment. So, when a command reuses another command's code body, the command can define its own command properties, for example,

privilege, help string, etc, but does not need to write Variable sub-segment and Code sub-segment.

An example is shown below. "cmd 2" reuses code body of "cmd 1" and meanwhile, "cmd 2" can has its own help strings and other command properties.

```
CMD_BEGIN
COMMAND   = test-cmd 1
FUNC_NAME = _cmd_1_cb
HELP      = Test command
HELP      = Command 1
VARIABLE_BEGIN
    ...
VARIABLE_END

CODE_BEGIN
    ...
CODE_END

CMD_END

CMD_BEGIN
COMMAND    = debug-cmd 2
FUNC_REUSE = _cmd_1_cb
HELP       = Debug command
HELP       = Command 2
CMD_END
```

PRIVILEGE =

(Mandatory)

Privilege level of the command, `ICLI_PRIVILEGE_XX`, defined in `icli_porting.h`. Current privileges are from `ICLI_PRIVILEGE_0` to `ICLI_PRIVILEGE_15`. The privilege is higher if the number is larger. The command can be executed only by the session with a higher or equal privilege.

For example, if the privilege of a command is `ICLI_PRIVILEGE_7`, then sessions with privilege in `ICLI_PRIVILEGE_0`~`ICLI_PRIVILEGE_6` can not access the command. The sessions with privilege in `ICLI_PRIVILEGE_7`~`ICLI_PRIVILEGE_15` can access the command.

PROPERTY =

(Optional)

Property of the command, `ICLI_CMD_PROP_XXXX`, defined in `icli_types.h`. You can use '|' to combine them. There are 8 command properties, where if the property is defined as the value of `0x00`, then this property is the default property.

```
#define ICLI_CMD_PROP_ENABLE    0x00
#define ICLI_CMD_PROP_DISABLE   0x01

#define ICLI_CMD_PROP_VISIBLE   0x00
#define ICLI_CMD_PROP_INVISIBLE 0x02

#define ICLI_CMD_PROP_GREP      0x04
#define ICLI_CMD_PROP_NOT_GREP  0x00

#define ICLI_CMD_PROP_LOOSELY   0x08
#define ICLI_CMD_PROP_STRICTLY  0x00
```

ICLI_CMD_PROP_ENABLE and ICLI_CMD_PROP_DISABLE are mutually exclusive. They indicate the command is executable or not. When the user inputs a valid command, ICLI engine will check this property to decide whether or not the corresponding command execution function is able to be executed. If it is ICLI_CMD_PROP_ENABLE, then the command execution function is able to be executed. If it is ICLI_CMD_PROP_DISABLE, then the command execution function will not be executed.

ICLI_CMD_PROP_VISIBLE and ICLI_CMD_PROP_INVISIBLE are mutually exclusive. These two indicate if the command is visible to users. In other words, when the user strokes TAB key or inputs '?' for help, ICLI engine checks this property to decide whether or not to display the command to the user. If the property is ICLI_CMD_PROP_VISIBLE, then the command will be displayed. If the property is ICLI_CMD_PROP_INVISIBLE, then the command will not be displayed to the user.

ICLI_CMD_PROP_GREP and ICLI_CMD_PROP_NOT_GREP are mutually exclusive. These two indicate if the command has the grep function to format the output. If the property is ICLI_CMD_PROP_GREP, then the command will have the grep function. If the property is ICLI_CMD_PROP_NOT_GREP, then the command will not have the grep function.

ICLI_CMD_PROP_LOOSELY and ICLI_CMD_PROP_STRICTLY are mutually exclusive. ICLI_CMD_PROP_LOOSELY indicates that the extra words are allowed after the complete valid command string. In this case, if the property is ICLI_CMD_PROP_STRICTLY, then the command string becomes invalid.

For example,

```
COMMAND   = ip dhcp snooping
...
PROPERTY  = ICLI_CMD_PROP_LOOSELY
...

CMD_END
```

Then if user inputs a command string "ip dhcp snooping is to enable ip dhcp snooping function", the command string is valid because "ip dhcp snooping" is valid and because of ICLI_CMD_PROP_LOOSELY, ICLI engine will ignore the extra string, "is to enable ip dhcp snooping function". But if the property is ICLI_CMD_PROP_STRICTLY, then this command string is invalid because ICLI engine still parse "is to enable ip dhcp snooping function" that is not a valid command.

If the tag value is ignored, the default value is 0 then the default property is

```
ICLI_CMD_PROP_ENABLE | ICLI_CMD_PROP_VISIBLE | ICLI_CMD_PROP_NOT_GREP | ICLI_CMD_PROP_STRICTLY
```

Please note that the value of ICLI_CMD_PROP_ENABLE is 0 and the value of ICLI_CMD_PROP_VISIBLE is also 0. So, if you want the command property ICLI_CMD_PROP_ENABLE | ICLI_CMD_PROP_VISIBLE | ICLI_CMD_PROP_GREP then you can simply define **PROPERTY =** ICLI_CMD_PROP_GREP.

CMD_MODE =

(Mandatory)

Command mode, ICLI_CMD_MODE_XXXX defined in icli_porting.h, that the command works at. The command mode is a method to categorize the commands. And, the command is visible and can be executed only in the work mode. So, the module designer must be aware of the deployment of his each command.

The advance feature is you can define a command in multiple command modes if the command works in multiple command modes.

Please see the example below.

```
COMMAND   = temperature { get | set <0-100> }
...
CMD_MODE  = ICLI_CMD_MODE_EXEC
CMD_MODE  = ICLI_CMD_MODE_GLOBAL_CONFIG
...

CMD_END
```

Now, we have two ways to reuse command, the first one is **FUNC_NAME**/**FUNC_REUSE**, the second one is multiple **CMD_MODE**.

If a command works in different command modes and all properties of the command are the totally same in these different command modes, i.e., same privilege, same help, same byword, etc, then you should use the second way. Otherwise, the first way is preferred.

GOTO_MODE =

(Optional)

The tag tells which command mode will be after executing this command. This tag is not only for command execution, but also for parsing. If this is not defined correctly, the parsing and execution will get problem. However, this should be automatically generated by the tag, **SUB_MODE**. If you want to add this manually, you may discuss with CP (cpwang@vitesse.com) first.

SUB_MODE =

(Optional)

Command mode, `ICLI_CMD_MODE_XXXX` defined in `icli_porting.h`, that the command will create and will go into after execution. If this is defined, the following tasks will be automatically implemented. In other words, all necessary tasks to create a sub mode will be done automatically through this tag.

1. Generate command in `ICLI_CMD_MODE_GLOBAL_CONFIG`.
2. Generate commands in all sub modes with the property of INVISIBLE.
3. Generate the following commands in this sub mode.
    - exit
    - end
    - help
    - do <line>

IF_FLAG =

(Optional)

This is for `#if` conditional flag to enclose the command. The tag value will be pasted behind `#if` directive. For example, `IF_FLAG = defined(VTSS)`, then it will be `#if defined(VTSS)`. So, if the command execution depends on some conditions, then you can use this to define the conditions. For example, if you want this command disabled, then you can define `IF_FLAG = 0` and the command will be enclosed by `#if 0`.

CMD_VAR =

(Optional)

C variable for the corresponding word of command string **COMMAND**, that is, this is one-to-one mapping to each word of command string. An example is as follows.

```
COMMAND = a [ b | c ] d
CMD_VAR = v1
CMD_VAR = v2
CMD_VAR = v3
CMD_VAR = v4
```

`v1` is for `a`, `v2` is for `b`, `v3` is for `c` and `v4` is for `d`.

But if you do not need command variables for `a` and `c`, then you can ignore the tag values, but the tags are still needed for keeping the correspondence.

```
COMMAND = a [ b | c ] d
CMD_VAR =
CMD_VAR = v2
CMD_VAR =
CMD_VAR = v4
```

`v2` is for `b` `v4` is for `d`.

The C variable will be auto-declared in command execution function of generated C file according to the word type in command string **COMMAND**. For the first example, because `a`, `b`, `c` and `d` are keywords, the corresponding C type is `BOOL`. The declarations in command execution function of generated C file are as follows.

```
BOOL   v1 = FALSE;
BOOL   v2 = FALSE;
BOOL   v3 = FALSE;
BOOL   v4 = FALSE;
```

For the second example, the declarations in command execution function of generated C file are as follows.

```
BOOL   v2 = FALSE;
BOOL   v4 = FALSE;
```

For the syntac Repeat(…), the variables will be generated automatically and mapping to the repeat command words.

For example,

```
COMMAND = a { {b|c|d} … }
CMD_VAR = a
CMD_VAR = b
CMD_VAR = c
CMD_VAR = d
```

The command is equivalent to `a {b|c|d} {b|c|d} {b|c|d}`. For the red command words, three C variables, `b_1`, `c_1` and `d_1`, are automatically generated by ICLI egnine for mapping. For the green command words, three C variables, `b_2`, `c_2` and `d_2`, are automatically generated by ICLI engine for mapping.

Through the **CMD_VAR**, you can know the input value of this command from the user and use these C variables in **CODE** sub-segment.

The C data type of command variable declared for each corresponding variable type is listed in Appendix A.

RUNTIME =

(Optional)

Callback for runtime check on the corresponding word of command string **COMMAND**, that is, this is one-to-one mapping to each word of command string. The mapping rule is the same with the rule for **CMD_VAR**.

The callback is invoked at the time of command execution, pressing TAB and '?'. It asks for present check, byword, help and runtime value range. The prototype of the callback is `icli_runtime_cb_t` described as the follows.

```
/*
    ICLI_ASK_PRESENT    : ask if the word is present or not
    ICLI_ASK_BYWORD     : ask byword, and this works on non-keyword
    ICLI_ASK_HELP       : ask help string
    ICLI_ASK_RANGE      : ask integer range for signed or unsigned,
                          this works on variables for all signed and
                          unsigned integer or integet list
    ICLI_ASK_PORT_RANGE : ask port type and list for the port range,
                          this works on <port_type_id>, <port_type_list>,
                          <port_type>, <port_id>, <port_list>
    ICLI_ASK_CWORD      : ask all possible customized words for <cword>
                          use 'NULL' for the end
    ICLI_ASK_VCAP_VR    : ask range for vcap_vr
*/
typedef enum {
    ICLI_ASK_PRESENT,
    ICLI_ASK_BYWORD,
    ICLI_ASK_HELP,
    ICLI_ASK_RANGE,
    ICLI_ASK_PORT_RANGE,
    ICLI_ASK_CWORD,
    ICLI_ASK_VCAP_VR,
} icli_runtime_ask_t;

typedef union {
    BOOL                        present;
    char                        byword[ICLI_RUNTIME_MAX_LEN + 4];
    char                        help[ICLI_RUNTIME_MAX_LEN + 4];
    icli_range_t                range;
    icli_stack_port_range_t     port_range;
    char                        *cword[ICLI_CWORD_MAX_CNT];
    icli_ask_vcap_vr_t          vcap_vr;
} icli_runtime_t;

/*
    INPUT
        session_id : session ID
        ask        : what is asked at runtime

    OUTPUT
        runtime
            ICLI_ASK_PRESENT    : runtime.present
            ICLI_ASK_BYWORD     : runtime.byword
            ICLI_ASK_HELP       : runtime.help
            ICLI_ASK_VALUE      : runtime.range
            ICLI_ASK_PORT_RANGE : runtime.port_range
            ICLI_ASK_CWORD      : runtime.cword
            ICLI_ASK_VCAP_VR    : runtime.vcap_vr

    RETURN
        TRUE
            ICLI engine will check the value in runtime.

            ICLI_ASK_PRESENT    : runtime.present == TRUE,  enable the word
                                  runtime.present == FALSE, disable the word
            ICLI_ASK_BYWORD     : use runtime.byword
            ICLI_ASK_HELP       : use runtime.help
```

```
            ICLI_ASK_VALUE      : use runtime.range
            ICLI_ASK_PORT_RANGE : use runtime.port_range
            ICLI_ASK_CWORD      : use runtime.cword
            ICLI_ASK_VCAP_VR    : use runtime.vcap_vr

        FALSE
            ICLI engine will ignore the value in runtime.

            ICLI_ASK_PRESENT    : the word is present
            ICLI_ASK_BYWORD     : use original one in *.icli
            ICLI_ASK_HELP       : use original one in *.icli
            ICLI_ASK_VALUE      : use original one in *.icli
            ICLI_ASK_PORT_RANGE : use system port range
            ICLI_ASK_CWORD      : <cword> works as <word>
            ICLI_ASK_VCAP_VR    : no range limit
*/
typedef BOOL (icli_runtime_cb_t)(
    IN  u32                 session_id,
    IN  icli_runtime_ask_t  ask,
    OUT icli_runtime_t      *runtime
);
```

So, you can use the runtime check in the following cases.

1. The word in command will be enabled or disabled at run time. For example, the corresponding component is enabled or disabled.

2. The range of value will be changed at run time. In this case, you can change the byname and help correspondingly to show the user.

For each ask type, it may works on some specific word types only.

ICLI_ASK_PRESENT	:	for keyword and all variable types
ICLI_ASK_BYWORD	:	for all variable types, but not for keyword
ICLI_ASK_HELP	:	for keyword and all variable types
ICLI_ASK_VALUE	:	only for the variable types of <range_lsit>, <int>, <uint>, <word>, <kword>, <string> and <line>.
ICLI_ASK_PORT_RANGE	:	for <port_type_id> and <port_type_list>.
ICLI_ASK_CWORD	:	only for <cword>.
ICLI_ASK_VCAP_VR	:	only for <vcap_vr>.

An example is as follows. RSTP may be enabled or disabled at run time, so the word "rstp" has a runtime check for it.

```
FUNCTION_BEGIN
static BOOL _rstp_runtime(
    IN    u32                   session_id,
    IN    icli_runtime_ask_t    ask,
    OUT   icli_runtime_t        *runtime
)
{
    switch ( ask ) {
    case ICLI_ASK_PRESENT:
        if ( rstp_enable() ) {
            runtime.present = TRUE;
        } else {
            runtime.present = FALSE;
        }
        return TRUE;

    default:
        break;
    }
    return FALSE;
```

```
}
FUNCTION_END

CMD_BEGIN
COMMAND   = stp mode [ mstp | rstp | mrstp ]
...
RUNTIME   =
RUNTIME   =
RUNTIME   =
RUNTIME   = _rstp_runtime
RUNTIME   =
...
CMD_END
```

By using `ICLI_ASK_PRESENT` on the first keyword of a command, this command can be enabled+visiable and disabled_invisible at runtime. The following example tells that the command is enabled and visible when STP mode is enabled. Otherwise, the command is hidden, that is disabled and invisible.

```
FUNCTION_BEGIN
static BOOL _stp_runtime(
    IN    u32                   session_id,
    IN    icli_runtime_ask_t    ask,
    OUT   icli_runtime_t        *runtime
)
{
    switch ( ask ) {
    case ICLI_ASK_PRESENT:
        if ( stp_enable() ) {
            runtime.present = TRUE;
        } else {
            runtime.present = FALSE;
        }
        return TRUE;

    default:
        break;
    }
    return FALSE;
}
FUNCTION_END

CMD_BEGIN
COMMAND   = stp mode [ mstp | rstp | mrstp ]
...
RUNTIME   = _stp_runtime
RUNTIME   =
RUNTIME   =
RUNTIME   =
RUNTIME   =
...
CMD_END
```

BYWORD =

(Optional)

Alternative word represents the corresponding word of command string when the user presses TAB key or '?'. Generally, the word displayed when pressing TAB or '?' is the word in **COMMAND**. If the corresponding **BYWORD** is defined, then this byword will be displayed, but not the word in **COMMAND**.

On the other hand, the **BYWORD** works on variable only, but not on keyword.

Remember that the byword also can be got at run time through **RUNTIME**.

An example is as follows.

Example 1 is without byword.

```
COMMAND   = temperature set <0-100>
...
BYWORD    =
BYWORD    =
BYWORD    =
...
CMD_END
```

Then, the user inputs as follows.

```
switch> temperature set ?

<0-100>
```

Example 2 is with a byword.

```
COMMAND   = temperature set <0-100>
...
BYWORD    =
BYWORD    =
BYWORD    = <CelsiusDegree>
...
CMD_END
```

Then, the user inputs as follows.

```
switch> temperature set ?

<CelsiusDegree>
```

HELP =

(Optional)

Help string for the corresponding word of command string.

This is displayed when pressing '?' to get the full descriptions for next possible command words.

If the help string is long, you may use '\' to concatenate lines.

Remember that the help also can be got at run time through **RUNTIME**.

An example is as follows.

```
COMMAND   = temperature { get | set <0-100> }
...
BYWORD    =
BYWORD    =
BYWORD    =
BYWORD    = <CelsiusDegree>
HELP      =
HELP      = get the current temperature in Celsius degree
HELP      =
HELP      = the range of degree is from 0 to 100
...
CMD_END
```

Then, the user inputs as follows.

```
switch> temperature ?

get                 get the current temperature in Celsius degree

set

switch> temperature set ?

<CelsiusDegree>     the range of degree is from 0 to 100
```

"set" does not have help string to display because the tag value of its corresponding "HELP" is empty. On the other hand, the BYWORD of "set" does not work because "set" is a keyword, not a variable.

MODE_VAR =

(Optional)

C variable for the variable in the mode entry command. This does not need to one-to-one mapping as **CMD_VAR** to **COMMAND** but ICLI engine will help to search the variable in the mode entry command and to map to it. The C variable will be auto-declared in command execution function of generated C file according to the variable type in the mode entry command.

```
MODE_VAR  = vid
```

The example is in Section 0 and the mode entry command is "`interface vlan <1-4094>`". ICLI engine will automatically find `vid` is for `<1-4094>`.

Therefore, we can get the VLAN ID through the C variable `vid`, and, in this example, use `vid` to set ip address on the correct.

VARIABLE_BEGIN

VARIABLE_END

(Optional)

Marks the beginning and end of variable sub-segment.

The sub-segment declares the variables that will be used in the following **Code** sub-segment and you may also initialize variables of **CMD_VAR** and **MODE_VAR** here. Please refer the example in Section 0.

CODE_BEGIN

CODE_END

(Optional)

Marks the beginning and end of code sub-segment.

This sub-segment contains the code body that implements the function of the command.

There is a C variable, `session_id`, which can be used in this segment. `session_id` is an input parameter of command execution function and identifies the current ICLI session.

On the other hand, you need `session_id` for the use of APIs exported in `icli_api.h` because it always is the first input parameter of APIs in `icli_api.h`. For example,

```
icli_session_printf(session_id, "test");
```

NO_FORM_DOC_CMD_DESC

NO_FORM_DOC_CMD_DEFAULT

NO_FORM_DOC_CMD_USAGE

NO_FORM_DOC_CMD_EXAMPLE

NO_FORM_VARIABLE_BEGIN

NO_FORM_VARIABLE_END

NO_FORM_CODE_BEGIN

NO_FORM_CODE_END

(Optional)

Their usages are the same with previous **DOC_CMD_DESC**, **DOC_CMD_DEFAULT**, **DOC_CMD_USAGE**, **DOC_CMD_EXAMPLE**, **VARIABLE_BEGIN**, **VARIABLE_END**, **CODE_BEGIN** and **CODE_END**. But, these tags defines the HTML descriptions and the code body for no form command.

If and only if there are codes in Code sub-segment, **NO_FORM_CODE_BEGIN** and **NO_FORM_CODE_END**, the no form command will be auto-generated by ICLI engine.

For the example below, "`no arp inspection`" will be auto-generated by ICLI engine and the duplicate words `arp inspection` will use the same command properties defined previously.

```
CMD_BEGIN
COMMAND   = arp inspection
...
HELP      = ARP configuration
HELP      = Arp Inspection configuration
...
VARIABLE_BEGIN
    ...
VARIABLE_END

CODE_BEGIN
    /* enable ARP inspection */
    ...
CODE_END

NO_FORM_CODE_BEGIN
    /* disable ARP inspection */
    ...
NO_FORM_CODE_END
CMD_END
```

DEFAULT_FORM_DOC_CMD_DESC

DEFAULT_FORM_DOC_CMD_DEFAULT

DEFAULT_FORM_DOC_CMD_USAGE

DEFAULT_FORM_DOC_CMD_EXAMPLE

DEFAULT_FORM_VARIABLE_BEGIN

DEFAULT_FORM_VARIABLE_END

DEFAULT_FORM_CODE_BEGIN

DEFAULT_FORM_CODE_END

(Optional)

Their usages are the same with previous **DOC_CMD_DESC**, **DOC_CMD_DEFAULT**, **DOC_CMD_USAGE**, **DOC_CMD_EXAMPLE**, **VARIABLE_BEGIN**, **VARIABLE_END**, **CODE_BEGIN** and **CODE_END**. But, these tags defines the HTML descriptions and the code body for default form command.

If and only if there are codes in Code sub-segment, **DEFAULT_FORM_CODE_BEGIN** and **DEFAULT_FORM_CODE_END**, the default form command will be auto-generated by ICLI engine.

For the example below, "`default arp inspection`" will be auto-generated by ICLI engine and the duplicate words `arp inspection` will use the same command properties defined previously.

```
CMD_BEGIN
COMMAND    = arp inspection
...
HELP       = ARP configuration
HELP       = Arp Inspection configuration
...
VARIABLE_BEGIN
    ...
VARIABLE_END

CODE_BEGIN
    /* enable ARP inspection */
    ...
CODE_END

DEFAULT_FORM_CODE_BEGIN
    /* reset ARP inspection to be default*/
    ...
DEFAULT_FORM_CODE_END
CMD_END
```

When deploying no/default form command, one thing must be sured is that the command syntaxes of normal command and no/default form command should be the totally same.

For example, the normal command of ARP inspection is "`arp inspection`" and its no form command is "`no arp inspection`". Except "`no`", they have the same syntax so it can use no form.

However, the normal command of IGMP is "`ip igmp {V1|V2|V3}`" and its no form command is "`no ip igmp`". The no form command does not have the syntax `{V1|V2|V3}`, their syntaxes are different so the no form can not be applied.

3.2.5 Defining Constant String

The constant string can be defined everywhere, but except the areas inside Include Segment, Function Segment, Variable Intial Sub-segment and Code Sub-segment. The reason is these areas are pasted exactly to generated C and H files so ICLI engine does not take care of any thing in these areas.

In other words, if there is a **TAG-VALUE** defined outside these areas and the tag is not reserved by ICLI engine, then the **TAG-VALUE** will be took as a constant string definition no matter it is upper or lower case. When it is referred, the prefix "##" is needed to indicate that the following string is a name of constant string.

This feature make the string easily reused, for example, **HELP** string.

For example,

```
SHOW_HELP_str = show system information

COMMAND   = show ip interface
...
HELP      = ##SHOW_HELP_str
HELP      =
HELP      =
...
CMD_END

COMMAND   = show mac address
...
HELP      = ##SHOW_HELP_str
HELP      =
HELP      =
...
CMD_END
```

On the other hand, no matter where the constant string is defined in the script, it can be used in whole scope in the script.

For example,

```
COMMAND   = show ip interface
...
HELP      = ##SHOW_HELP_str
HELP      =
HELP      =
...
CMD_END

SHOW_HELP_str = show system information

COMMAND   = show mac address
...
HELP      = ##SHOW_HELP_str
HELP      =
HELP      =
...
CMD_END
```

For example,

```
COMMAND   = show ip interface
...
HELP      = ##SHOW_HELP_str
HELP      =
HELP      =
...
CMD_END

COMMAND   = show mac address
...
HELP      = ##SHOW_HELP_str
```

```
HELP      =
HELP      =
...
CMD_END

SHOW_HELP_str = show system information
```

In the previous two examples, `SHOW_HELP_str` is defined at the middle and last line, but it still can be used in the commands.

3.2.6 An Example

This example uses '!' for line comment and is described as follows.

```
! Begin of Command segment
CMD_BEGIN

! command description
DOC_CMD_DESC    = this command is used to set management ip interface.
! default value
DOC_CMD_DEFAULT = the default IP is 192.168.0.1/255.255.255.0
! usage description
DOC_CMD_USAGE   = when the default IP does not work on your network,\
                  you can use this command to modify the IP for your network.
! example
DOC_CMD_EXAMPLE = if you wants to set ip and netmask to be 10.1.1.1/255.0.0.0,\
                  you can execute the command as follows.
DOC_CMD_EXAMPLE = Switch# ip address 10.1.1.1 255.0.0.0

! Command string
COMMAND   = ip address <ipv4_ucast> <ipv4_netmask>
! Privilege level of the command
PRIVILEGE = ICLI_PRIVILEGE_8
! Command property
PROPERTY  = ICLI_CMD_PROP_ENABLE | ICLI_CMD_PROP_VISIBLE

! C variable for "ip"
CMD_VAR   =
! C variable for "address"
CMD_VAR   =
! C variable for "<ipv4_ucast>"
CMD_VAR   = ip
! C variable for "<ipv4_netmask>"
CMD_VAR   = netmask

! Help string for "ip"
HELP      = ip interface
! Help string for "address"
HELP      = ip address set
! Help string for "<ipv4_ucast>"
HELP      = unicast IP address
! Help string for "<ipv4_netmask>"
HELP      = netmask
```

```
! The command works at interface VLAN mode.
CMD_MODE = ICLI_CMD_MODE_INTERFACE_VLAN

! C variable for "<1-4094>"
MODE_VAR  = vid

! Variable declaration and initialization
VARIABLE_BEGIN
    char ip_str[20];
    char netmask_str[20];
VARIABLE_END

! Command implementation body
CODE_BEGIN
    //translate IP and netmask to string format
    icli_ipv4_to_str(ip, ip_str);
    icli_ipv4_to_str(netmask, netmask_str);
    ICLI_PRINTF("Set %s/%s on VLAN %d successfully\n",
            ip_str, netmask_str, vid);
CODE_END

! End of Command segment
CMD_END
```

4 Make on eCos

4.1 Make File

To automatically generate and register ICLI commands, you can simply copy the following two lines into your component make file, said, **`module_x.in`**.

```
---------------------------------------------------------------------------------------------------

# Built-in ICLI

$(eval $(call add_icli,$(foreach m, x_0 x_1,$(DIR_x_script)/$(m).icli)))

---------------------------------------------------------------------------------------------------
```

where `x_0` and `x_1` are your ICLI script files, `x_0.icli` and `x_1.icli`.

`DIR_x_script` is the directory where `x_0.icli` and `x_1.icli` are stored.

A Appendix: Variable Types

This appendix lists all variable types supported by ICLI. "**C Type**" in the description means the data type will be declared by ICLI in generated C file. So that you can access the variable correctly in C. "**Description**" describes the legal syntax of the variable type. "**Legal Input**" shows the examples legal to the variable type. "**Illegal Input**" shows the examples illegal to the variable type.

A.1 <*a*~*b*>

C Type

If with negative, icli_signed_range_t *

if without negative, icli_unsigned_range_t *

Description

A list of range in the range of *a* and *b*, where *a* < *b*, the max number of range blocks in the list is 8. An example is <-9~90>.

And, *a* and *b* can be digial numbers or constant. If it is a constant then it needs to be enclosed by single quote. An example is <1~'ACL_MAX_CNT'>.

Legal Input

-5

0,0,0,0

-5--3,-1-0,6,10-90

1,3,4,6,7,8,5,2

Illegal Input

-30

-5--3,-1-0,6,10-99

-10,2,3,4,5,6,7,8,9

A.2 <*a*-*b*,*c*,*d*-*e*>

C Type

u32

Description

Integer in a range. The range is separated by ',' and each block between ',' can be a single decimal interger or a range value. The range value uses '-' to indicate the range. In this case, *a*, *b*, *c*, *d* and *e* are decimal integers and *a* < *b* and *d* < *e*. The maximum number of range blocks is 8.

Assume, "-5--3,20,-1-0,6,15-119,-4" has 6 range blocks.

So the legal input value should be either (>= -5 && <= -3) or (>= -1 && <=0) or (==6) or (>=15 && <= 119).

Except to be digital numbers, *a* and *b* can also be constants. If it is a constant then it needs to be enclosed by single quote. An example is <1-‘ACL_MAX_CNT’>, or <’MIN_NUM’-‘MAX_NUM’>

Legal Input

-4

0

6

17

118

Illegal Input

-6

-2

3

11

120

A.3 <clock_id>

C Type

icli_clock_id_t

Description

The ID is an array of hex value in 8 bytes.

Legal Input

00:01:02:03:04:05:06:07

0-1-2-3-4b-5-6-7

0-001-2-3:4:5:006:7a

Illegal Input

0:1:2:3:4:5:6:

0-111-2-3-4-5-6-7

0-1-2-3:4:5-6.7

A.4 <cword>

C Type

char *

Description

Constant words defined in runtime

Legal Input

Words defined in runtime

Illegal Input

words not defined

A.5 <date>

C Type

icli_date_t

Description

Date in yyyy/mm/dd, yyyy=1970-2037, mm=1-12, dd=1-31

Legal Input

1970/09/25

2011/05/26

2037/12/31

Illegal Input

1969/09/25

2011/050/26

203a/12/31

A.6 <domain_name>

C Type

char *

Description

Domain name compliant with RFC1123.

<domain> ::= <subdomain>

<subdomain> ::= <label> | <subdomain> "." <label>

<label> ::= <let-dig> [[<ldh-str>] <let-dig>]

<ldh-str> ::= <let-dig-hyp> | <let-dig-hyp> <ldh-str>

<let-dig-hyp> ::= <let-dig> | "-"

<let-dig> ::= <letter> | <digit>

<letter> ::= A-Za-z

<digit> ::= 0-9

<domain_name*m*> and <domain_name*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

abc.1.com

1.2.3.4

1.2-3

Illegal Input

abc.1.

1.2.3.4.

1.2-3-

A.7 <dpi>

C Type

u8

Description

Drop Precedence Level. If the platform is Juguar1, the range is 0 to 3.

Otherwise, the range is 0 to 1.

Legal Input

0

1

2 (for Juguar1)

3 (for Juguar1)

Illegal Input

-0

-1

4

5

A.8 <dscp>

C Type

u8

Description

It provides specific DSCP PHBs and the valid PHbs are be, af11, af12, af13, af21, af22, af23, af31, af32, af33, af41, af42, af43, cs1, cs2, cs3, cs4, cs5, cs6, cs7, ef, va.

Legal Input

b

be

af13

cs6

e

ef

Illegal Input

be1

af1

c

cs

vb

A.9 <dword>

C Type

char *

Description

A single word with all characters in numeric letter, 0-9.

<dword*m*> and <dword*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

0203

1235

10000

Illegal Input

f12345

-123

_iso34

A.10 <fword>

C Type

char *

Description

A single word with a floating point.

<fword*m*> and <fword*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

0.203

12.35

1000.0

Illegal Input

f2345

-123

100.00.0

A.11 <hexval>

C Type

icli_hexval_t

Description

A hex value begins with '0x' or '0X' and its default maximum length is 128 bytes.

<hexval*m*> and <hexval*m-n*> are supported, where *m* is the maximum length of the value and *n* is the minimum length.

Legal Input

0x012345678

0X567890abcdfe

Illegal Input

12345678

0X567890k

A.12 <hhmm>

C Type

icli_time_t

Description

Time in hh:mm, hh=0-23, mm=0-59

Legal Input

00:00

05:05

23:15

Illegal Input

:00:00

05:05:

23::15

24:15:59

A.13 <host_name>

C Type

char *

Description

<host_name> ::= <let-dig> [[<ldh-str>] <let-dig>]

<ldh-str> ::= <let-dig-hyp> | <let-dig-hyp> <ldh-str>

<let-dig-hyp> ::= <let-dig> | "-"

<let-dig> ::= <letter> | <digit>

<letter> ::= A-Za-z

<digit> ::= 0-9

The default maximum length is 63 bytes.

<host_name*m*> and <host_name*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

Abc0998

9

12-34

67-gh

Illegal Input

8@

2.4

6-8.com

A.14 <int>

C Type

i32

Description

32-bit signed integer, the range is -2147483648 to 2147483647.

This has RUNTIME.range feature.

Legal Input

-100

-0

1234567890

Illegal Input

-2147483649

2147abc

21474836470

A.15 <int16>

C Type

i16

Description

16-bit signed integer, the range is -32768 to 32767.

Legal Input

-100

-0

23456

Illegal Input

-2147449

7abc

14748364

A.16 <int8>

C Type

i8

Description

8-bit signed integer, the range is -128 to 127.

Legal Input

-100

-0

123

Illegal Input

-83649

21bc

128

A.17 <ipv4_abc>

C Type

vtss_ip_t

Description

IPv4 IP address in the format of ddd.ddd.ddd.ddd where 'd' is a decimal digit and the address must be in class A, B or C. If it is in class D or E, then it is invalid.

Legal Input

1.0.0.0

10.254.255.0

223.255.255.255

Illegal Input

224.0.0.0

240.255.255.254

255.0.0.1

A.18 <ipv4_addr>

C Type

vtss_ip_t

Description

Any IPv4 IP address in the format of ddd.ddd.ddd.ddd where 'd' is a decimal digit and the range of each ddd is 0 – 255.

Legal Input

0.0.0.0

1.2.3.4

255.255.255.255

Illegal Input

0.0.0

0.0.0.256

1.2.3:4

A.19 <ipv4_mcast>

C Type

vtss_ip_t

Description

Multicast IPv4 IP address in the format of ddd.ddd.ddd.ddd where 'd' is a decimal digit. It is in the range of 224.0.0.0 to 239.255.255.255.

Legal Input

224.0.0.0

230.254.255.0

239.255.255.255

Illegal Input

0.0.1.2

223.255.255.254

240.0.0.1

A.20 <ipv4_netmask>

C Type

vtss_ip_t

Description

IPv4 Netmask in the format of ddd.ddd.ddd.ddd where 'd' is a decimal digit.

Legal Input

0.0.0.0

255.255.255.0

255.252.0.0

Illegal Input

0.0.255.255

255.250.255.0

255.252.0.1

A.21 <ipv4_nmcast>

C Type

vtss_ip_t

Description

Non-multicast IPv4 IP address in the format of ddd.ddd.ddd.ddd where 'd' is a decimal digit. And, the IP address is the address not in the range of 224.0.0.0 to 239.255.255.255.

Legal Input

0.0.1.255

223.255.255.254

240.0.0.0

Illegal Input

224.0.0.0

235.0.1.255

239.255.255.255

A.22 <ipv4_prefix>

C Type

u32

Description

IPv4 prefix length, in the format of /n, when 'n' is in the range of 0 and 32.

Legal Input

/0

/32

/0010

Illegal Input

/-1

/33

/00101

A.23 <ipv4_subnet>

C Type

icli_ipv4_subnet_t

Description

IPv4 Subnet address in 2 formats, where 'd' is a decimal digit.

IP/netmask: ddd.ddd.ddd.ddd/ddd.ddd.ddd.ddd

IP/prefix-length: ddd.ddd.ddd.ddd/**dd**

Where IP is unicast IP address and prefix length **dd** is from 0 to 32.

Legal Input

10.1.1.1/255.0.0.0

10.1.1.1/0

10.1.1.1/32

223.255.255.254/8

Illegal Input

10.1.1.1/255.0.1.0

10.1.1.1/-1

10.1.1.1/33

224.1.1.1/8

A.24 <ipv4_ucast>

C Type

vtss_ip_t

Description

Unicast IPv4 IP address in the format of ddd.ddd.ddd.ddd where 'd' is a decimal digit.

The following 3 cases are illegal:

1. In the range of 224.0.0.0 to 239.255.255.255,

2. ddd.ddd.ddd.0,

3. ddd.ddd.ddd.255.

Legal Input

0.0.1.2

223.255.255.254

240.0.0.1

Illegal Input

0.0.1.255

224.0.0.1

240.0.0.0

A.25 <ipv6_addr>

C Type

vtss_ipv6_t

Description

Any IPv6 address, in the format of hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh, where 'h' is a hex digit. "::" can be used to skip some hex digits, but it can happen once only.

Legal Input

::

::1

0:0:0:0:0:0:0:0

1234::5678:90ab

1234::

::5678:90ab

Illegal Input

:

:::1

0:0:0:0:0:0:0:0:0

1234::5678::90ab

1234:::5678:90ab

1234::5678:90ab:

A.26 <ipv6_mcast>

C Type

vtss_ipv6_t

Description

Multicast IPv6 address, in the format of **hh**hh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh, where 'h' is a hex digit. "::" can be used to skip some hex digits, but it can happen once only.

But, the first **hh** must be 0xFF.

Legal Input

FF00::

FF55::1

FF34::cd:82

FF81:5678::ab

Illegal Input

::

FF0::

3F55::1

1234::cd:90ab

::5678:90ab

A.27 <ipv6_netmask>

C Type

vtss_ipv6_t

Description

Any IPv6 netmask, in the format of hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh, where 'h' is a hex digit. "::" can be used to skip some hex digits, but it can happen once only.

Legal Input

FF00::

FFF0::

FFFF:FF00::

FFFF:FFFF:FC00::

Illegal Input

::FF00

FF10::

FFFF::FF00::

:FFFF:FFFF:FC00::

A.28 <ipv6_prefix>

C Type

u32

Description

IPv6 prefix length, in the format of /n, when 'n' is in the range of 0 and 128.

Legal Input

/0

/128

/0000101

Illegal Input

/-1

/129

/0001011

A.29 <ipv6_subnet>

C Type

icli_ipv6_subnet_t

Description

IPv6 Subnet address, IP/Netmask or IP/mask-bits

Legal Input

::/FF00::

::1/FFFF:FF00::

0:0:0:0:0:0:0:0/33

1234::5678:90ab/1

1234::/128

::5678:90ab/F000::

Illegal Input

::/FF0::

::1/FFFF:FF0::

0:0:0:0:0:0:0:0/133

1234::5678:90ab/200

1234::/228

::5678:90ab/F000:F:

A.30 <ipv6_ucast>

C Type

vtss_ipv6_t

Description

Unicast IPv6 address, in the format of **hh**hh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh, where 'h' is a hex digit. "::" can be used to skip some hex digits, but it can happen once only.

But, the first **hh** must NOT be 0xFF.

Legal Input

::

FF0::

3F55::1

1234::cd:90ab

::5678:90ab

Illegal Input

FF00::

FF55::1

FF34::cd:82

FF81:5678::ab

A.31 <kword>

C Type

char *

Description

A single word, but must begin with an alphabet, A-Z or a-z, not a numeric letter.

<kword*m*> and <kword*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

ab2c3

f123

iso3456

Illegal Input

12345

f 123

_iso34

A.32 <line>

C Type

char *

Description

Any string that may contains several words with any characters separated by spaces.

<line*m*> and <line*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

This is a book

"This is a book"

123 45

"123 45"

12345

"12345"

a 123

"a 123"

Illegal Input

<enter>

A.33 <mac_addr>

C Type

vtss_mac_t

Description

Any Ethernet MAC address. hh:hh:hh:hh:hh:hh where 'h' is a hex digit.

Legal Input

00:11:2a:bc:de:f9

0:1:0:46:5:3a

Illegal Input

00:11:2a:bc

0.1.0.46.5.3a

A.34 <mac_mcast>

C Type

vtss_mac_t

Description

Multicast MAC address

hh:hh:hh:hh:hh:hh where **hh** & 0x01 == 0x01.

Legal Input

02:11:2a:bc:de:f9

0:1:0:46:5:3a

Illegal Input

03:11:2a:bc:45:fd

b:1:0:46:5:3a

A.35 <mac_ucast>

C Type

vtss_mac_t

Description

Unicast MAC address

hh:hh:hh:hh:hh:hh where **hh** & 0x01 != 0x01.

Legal Input

02:11:2a:bc:de:f9

0:1:0:46:5:3a

Illegal Input

03:11:2a:bc:45:fd

b:1:0:46:5:3a

A.36 <oui>

C Type

icli_oui_t

Description

Organizationally Unique Identifier (OUI) is a 24-bit number that is the first three bytes of MAC address. And, this should be unicast.

Legal Input

00-01-02

14:05:07

ba:09:fe

Illegal Input

01-01-02

14:05:07:

Ba::fe

A.37 <pcp>

C Type

icli_unsigned_range_t *

Description

Priority Code Point.

The valid inputs are specific (0, 1, 2, 3, 4, 5, 6, 7) or range (0-1, 2-3, 4-5, 6-7, 0-3, 4-7) or any(0-7)

Legal Input

5

2-3

6-7

0-7

Illegal Input

8

2-4

6-8

5-7

A.38 <port_type_id>

C Type

icli_switch_port_range_t

Description

The format is “port_type switch_id/port_id”.

Legal Input

Gi 1/1

Fast 1/15

10gi 2/23

Illegal Input

gi 1/0

gi1/1,2

gi 1/2-5

tengi 2/23

A.39 <port_type_list>

C Type

icli_stack_port_range_t *

Description

The format is “port_type switch_id/port_list”.

Legal Input

gi 2/13

gi 2/13,13 10gi 1/1

fast 1/3-6,9,7 gi 1/5

2.5G 1/5 fast 1/7 gi 1/3

5G 1/4,6;2/9,17-23 gi 1/1,3,5 fast 1/2,4-8;3/5,7

Illegal Input

gi 0/25

fast 2/13,

tengi 1,2/

gi 1,2,3/1,3-6,2

tengi 1/4,6,2;2/9,17-23

A.40 <range_list>

C Type

icli_range_t *

Description

A list of range, the max number of range blocks in the list is 8. The input allows incremental only, not decremental and not equal.

This has RUNTIME.range feature.

Legal Input

-5

-5--3,-1-0,6,10-99

1,2,3,4,5,6,7,8

Illegal Input

-5--30

-5--3,-7,6-10

1,1,1

1,2,3,4,5,6,7,8,9

A.41 <string>

C Type

char *

Description

A string may contain several words separated by spaces, must enclosed in double-quote “”.

<string*m*> and <string*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

“123 45”

“12345”

“a 123”

Illegal Input

“123 45

123 45”

123 45

A.42 <switch_id>

C Type

u32

Description

A single switch ID that will be checked according to the current switch configuration.

Assume currently there are switch 1,3,9.

Legal Input

1

3

9

Illegal Input

0

2

8

A.43 <switch_list>

C Type

icli_unsigned_range_t

Description

A list of switch IDs that will be checked according to the current switch configuration.

Assume currently there are switch 1,2,3,5,8,9.

Legal Input

2

2-3,9,8

1-3,5,5,8-9

Illegal Input

4

2-5,8-9

1-3,5-9

A.44 <time>

C Type

icli_time_t

Description

Time in hh:mm:ss, hh=0-23, mm=0-59, ss=0-59

Legal Input

00:00:00

05:05:05

23:15:59

Illegal Input

:00:00

05:0a:05

23:15:59:

24:15:59

A.45 <uint>

C Type

u32

Description

32-bit unsigned integer, the range is 0 to 4294967295.

This has RUNTIME.range feature.

Legal Input

0

100

4294967295

Illegal Input

-0

-1

10c

4294967296

A.46 <uint16>

C Type

u16

Description

16-bit unsigned integer, the range is 0 to 65535.

Legal Input

0

100

29496

Illegal Input

-0

-1

10c

429496

A.47 <uint8>

C Type

u8

Description

8-bit unsigned integer, the range is 0 to 255.

Legal Input

0

100

254

Illegal Input

-0

-1

10c

294

A.48 <url>

C Type

char *

Description

RFC-3986

<protocol>://[<username>[:<password>]@]<host>[:<port>][/<path>]

Requirements:

<protocol>: The scheme of URI. The input string allows the lowercase letters only and its maximum length is 31. It should be aware of tftp/ftp/http/https/file.

<username>: (Optional) User information. The maximum length is 63.

<password>: (Optional) User information. The maximum length is 63.

<host>: It is a domain name or an IPv4 address. The maximum length is 63.

<port>: (Optional) port number.

<path>: If the path is presented, it must separated by forward slash(/). The maximum length is 255.

<url*m*> and <url*m*-*n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

http://user@abc.com

tftp://1.2.3.4:123

ftp://vitesse.com:123/path

https://user:passwd@host.com.tw:123/path

file:///folder/path

Illegal Input

httpx://user@abc.com

tftp://1.2.3.4:123456

ftp://vitesse.com:123path

https://user:passwd@host.com.tw:123/path//

A.49 <vlan_id>

C Type

u32

Description

VLAN ID, an unsigned integer in the range of min_vlan_id to max_vlan_id, where min_vlan_id and max_vlan_id is configurable at run time.

By default, the valid range is 1-4095.

Legal Input

1

9

24

4094

Illegal Input

-5

0

4095

10000

A.50 <vlan_list>

C Type

icli_unsigned_range_t *

Description

VLAN ID List, a list of range blocks that must be from min_vlan_id to max_vlan_id.

By default, the valid range is 1-4095.

Legal Input

1100

1,1,200,300,55

1,20-30,90-2300

50,109,1010-4000

Illegal Input

-1100

1,200,30

1,20-30,9-2300

50,19,1010-4000

A.51 <vword>

C Type

char *

Description

A single word that allows 0-9a-zA-Z, but it does not allow all in 0-9 only.

<vword*m*> and <vword*m-n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

A123

123a

avcd

Illegal Input

123

-123

abcd+def

A.52 <word>

C Type

char *

Description

A single word without space inside.

<word*m*> and <word*m*-*n*> are supported, where *m* is the maximum length of the word and *n* is the minimum length.

Legal Input

12345

ab2c3

_isoi34*(

Illegal Input

123 45

ab 2c3

_iso i34*(

B Appendix: FAQ

B.1 Duplicate word

Because the ICLI parsing thinks the design is from left to right, the limitation on ICLI engine parsing is it does not allow the same words that overlap in mandatory{} or optional().

For example (1), 2 commands,

```
snmp client version <uint>

snmp { server | client } address <ipv4_ucast>
```

Then, the keyword '`client`' is overlapped illegally because '`client`' is in mandatory{} in the second command.

The example (2), 2 commands,

```
snmp client version <uint>

snmp [client] address <ipv4_ucast>
```

Then, the keyword '`client`' is still overlapped illegally because '`client`' is in optional[] in the second command.

The solution is as follows.

Example (1) is re-designed to the following 2 commands.

```
snmp client { version <uint> | address <ipv4_ucast> }

snmp server address <ipv4_ucast>
```

Example (2) is re-designed to the following 2 commands.

```
snmp client { version <uint> | address <ipv4_ucast> }

snmp address <ipv4_ucast>
```

In other words, the concept is to use the same flat prefix as most as possible. "`snmp client`" is the same flat prefix in the examples.

B.2 Multiple Optional Begin

If the command has the following designs, then the keyword '`x`' is multiple optional begin.

```
a [[x]]
```

or

```
a [ b | [x] ]
```

To solve this problem, a keyword is added before `[x]`. For example,

```
a [ k [x] ]
```

or

```
a [ b | k [x] ]
```

C Appendix: Short-cut keys

Function	***Shortcut key***	***Description***
Line scroll	Yes	If the length of command exceeds the width of windows, then "$" will be used to scroll the command input, but not display to the next line.
Cursor move	LEFT	Back one character to left.
	RIGHT	Forward one character to right.
	HOME	Go to the beginning of the line.
	Ctrl-a	
	END	Go to the end of the line.
	Ctrl-e	
Delete	Ctrl-n	The current line.
	DEL	A character at the cursor.
	Ctrl-d	
	Backspace	A character to the left of the cursor.
	Ctrl-h	
	Ctrl-u	All characters from the cursor to the beginning of the line.
	Ctrl-x	
	Ctrl-k	All characters from the cursor to the end of the line.
	Ctrl-w	A word from the cursor to the beginning of the word.
Page More	Spacebar	Next page.
	All other keys	
	Enter	Next line.
	Ctrl-c	Exit from more output.
	q	

	g	Go to the last line.
History	UP	Previous line.
	Ctrl-p	
	DOWN	Next line.
Context-Sensitive Help	TAB	• Unique : complete the token. • Ambiguous : list all possible tokens. • Behind prompt : list all possible tokens.
	?	The first '?' lists all possible tokens with help descriptions. The second immediate '?' lists all possible full commands.
	Ctrl-q	Display full command syntax.
Command Mode	Ctrl-z	Go back to Exec mode directly.