CLICON db2txt formatting

Olof Hagsand and Benny Holmgren

Aug, 2013

1 Introduction

CLICON is a software suite for configuration management including CLI generation, netconf interface, embedded databases and a backend plugin mechanism.

The CLICON db2txt formatting language is designed to allow a developer to format arbitrary text output based on content of the CLICON databases.

The formatting language is typically used to generate custom configuration output or to generate configuration files for various system daemons controlled by backend plugins.

2 Using db2txt from C-code

The db2txt parser can be called using one of two functions. One parses a file while the other parses a character string in memory. They both return a dynamically allocated string containing the output format.

Both functions take three arguments.

1. *clicon_handle* - The handle of the CLICON instance.
2. *db* - The filename of the database to use for variable substitutions.
3. *file* or *buf* - Either the name of the file to be parsed or a character string buffer.

Function prototypes:

```
char *clicon_db2txt(clicon_handle h, char *db, char *file);
char *clicon_db2txt_buf(clicon_handle h, char *db, char *buf);
```

3 Format overview

The format is plain text with variable substitution and inline code segments implementing loops and if-statements. Text is mapped directly from the input format to the output format, interpreting variables and code sections on the fly.

Inside code statements, input is interpreted differently and parsed similarly to a programming language in the sense that whitespace is ignored and there is a certain grammar expected.

An example of an input format

```
#
# This is a comment.
#
hostname $system.hostname->hostname
!
@if($ipv4.forwarding->status == (int)1)
ip forwarding
@else
no ip fowarding
@end
@each($interface[], $if)
interface $if->name
@each($if.inet.address[], $addr, " ip address \$addr")
@end
!
```

4 Special characters and keywords

The parser has a few special characters and keywords.

- *$* - marks the beginning of a variable substitution
- *\n* - substitutes into a newline
- *\r* - substitutes into a carriage return
- *\t* - substitutes into a tab
- *@set* - a function to set a variable
- *@each* - a loop function for database vectors
- *@if* - start of a conditional statement
- *@else* - alternative action of @if statement if false
- *@end* - end of @if or @each code sections
- *@include* - include a file to the format

It is possible to escape the characters and keywords using a backslash. As an example ”the price is \$100” would print the literal text rather then trying to substitute a variable named ’100’.

5 Variables and Callbacks

There are two types of actual variables, *parser* variables and *database* variables; and one kind of *callback* variables. Callback variables are a way for the parser to call a function in an application plugin for a substitution string.

A variable is noted by a leading *$* followed by the variable body. The format of the body depends on the type of variable as described below. A variable body can be surrounded by curly braces to explicitly specify the end of the body.

5.1 Parser variables

Parser variables only have relevance in the context of the db2txt parser, either in the parser's global context or within the limited context of a loop segment. A slight exception to that rule are CLICON application options defined in the *clicon.conf* file or programatically via the CLICON options API. You can substitute these options as variables in your format.

A parser variable must begin with a letter and followed by zero or more letters, digits or underscore.

Examples of parser variables follows.

```
$interface
$my_variable01
${theVariable}
$CLICON_CLI_MODE
```

5.2 Database variables

Database variables are stored in a database file as per the application's data specificaton. The variable body is noted by the specifying the database key in which the variable is stored, an arrow and then the variable name within that database key.

Database keys have the format of one or more node names delimited by dots. Node names and variable names must begin with a letter and followed by zero or more letters, digits or underscore. Nodes may also contain a dash.

The following example refers to the variable *host_name* in the database key *system.hostname.*

```
$system.hostname->host_name
  or
${system.hostname->host_name}
```

5.3 Callback variables

Callback variables is a callback from the parser to the application which can return an arbitrary string to be substituted into the formatted output.

Note that the returned string must be dynamically allocated as CLICON will free it once used.

The format of the callback variable body is an optional plugin name followed by two colons, then the function name followed by two parenthesis. If no plugin name is given, the function must exist in the master plugin whose symbols are visible in the global namespace.

Examples of callbacks follow.

Call the function *myfunc* in the master plugin:

```
$myfunc()
  or (assuming the master plugin is called 'master')
$master::myfunc()
```

Call the function *get_time()* in a plugin named *genconf*:

```
$genconf::get_time()
```

An optional argument can be passed to the callback function by either specifying a database or parser variable between the parenthesis. It is also possible to spcify the argument statically as a typed value. See section 6 for more information in typed values.

Some examples.

```
  $ntp::get_time($timezone)
  $internet::whoami((ipv4prefix)192.168.23.0/24)
  $random::seed((int)23)
  $customer::address((string)"John Doe")
```

The C-function prototype of the callback in the plugin looks as follows.

```
  typedef char *(clicon_db2txtcb_t)(cg_var *);
```

An example callback function:

```
char *my_callback(cg_var *arg)
{
    return strdup("Hello world!");
}
```

6 Typed values

In various code statements, a manually defined *value* may be specified as an alternative to variables. This is done by specifying the type and a text represe-nation of the value.

The types are based on the CLIgen datatypes and are specified within parenthesis. For example *(string)* or *(ipv4addr)*.

The value is noted as the text representation of the actual value. For example:

```
(int)42
(string)"the string"
(ipv4pfx)192.168.100.2/24
(url)http://www.acme.com/info/index.html
```

Available datatypes	
int	32-bit signed integer
long	64-bit signed integer
bool	1-bit boolean value
string	Double quoted character string
ipv4addr	IPv4 Address: 1.2.3.4
ipv4pfx	IPv4 Prefix: 1.2.3.4/34
ipv6addr	IPv6 Address: 2001:0db8:85a3:0042::::7334
ipv6pfx	IPv6 Prefix: 2001:0db8:85a3:0042::::7334/48
macaddr	MAC address f0:de:f1:1b:10:47
url	<proto>://[<user>[:<passwd>]@]<addr>[/<path>]
uuid	Universally Unique Identifier
time	ISO 8601 date+time: 2008-09-21T18:57:21.003

7 Code statements and sections

Code statemnts and sections provide functions and flow control to the formatting. A code statement starts with a @ character followed by a keyword. The keywords are case insensitive and the following are supported.

- *@set*
- *@if*
- *@else*
- *@end*
- *@each*
- *@include*

Code statements need to be followed by arguments or options enclosed in parenthesis. The exceptions are @else and @end which are not really statements but are just marking the end of a code section.

Note that a newline directly following a code statement is ignored. The reason is so that the code statements will not generate a lot of empty lines.

Consider the following example.

```
@if($system->name)
hostname $system->name
@end
```

If the newlines after the *@if* and *@end* statement would be included in the output format, it would consist of the hostname, if existing, with a newline before and after.

An effect of ignoring these newlines is that in-line formats need to include an explicit *n* if a newline is required. For example.

```
@if($system->name ? "hostname $system->name\n")
```

Code sections are normal text formatting between a @if or @each statement and the corresponding @end.

7.1 Set statement

The @set statement is used to set a parser variable. The syntax is *@set($name, <value>)* where *name* is the name of the variable to set and *<value>* can be a variable (any of the three types) or a typed value.

Examples.

```
@set($company, (string)"Acme Inc")
@set($date, $ntp::get_date())
@set($hostname, $system.hostname->host_name)
```

7.2 If statement

The @if statement allows for conditional text output. The syntax is as follow where the @else section is optional.

```
@if(<condition>)
 ... format if true ...
@else
 ... format if false ...
@end
```

The *condition* syntax is

```
<term> [<operator> <term>]
```

where *term* can be a parser variable, a database variable, a typed value or the *nil* keyword.

The condition can consist of either a single term or a two terms with an operator inbetween.

If only one term is given it will evaluate to true if the value is not equal *nil* or *(int)0*.

If two terms are specified, their values are compared based on the *operator* specified.

Valid operators	
==	Equal
!=	Not equal
<	Less than
>	Greater than
<=	Less or equal
>=	Greater or equal

The conditional statement will parse the format until the corresponding *@end* statement if the condition is true. An *@else* statement can be specified between the *@if* and *@end* in which case the formatting after *@else* is parsed in the condition is false.

The parser also supports the ternary operator with the following syntax.

```
@if(<condition>) ? <true-format> [ : <false-format>])
```

The *true-format* and the optional *false-format* are double-quoted strings and provide an way to specify in-line the formats to be parsed if the condition is true or false respectively.

7.3 Each statement

Looping through vectors is done using the *@each* statement. It will iterate through each entry in a vector key and evaluate the format for each entry. The format can either be specified in a code section after the *@each* statement until the corresponding *@end* or in-line as the third argument.

The format itself can contain new code statements such as *@if* or *@each* making conditional output and new loops possible within a loop.

The *each* syntax is:

```
@each(<vector-key>), <loop-variable> [ , <format> ])
```

The *vector-key* is a noted as a variable ending with two square brackets and specifies a database vector key. For example *$system.interface[]* which iterates through the list of database keys of the database vector 'system.interface[]'.

loop-variable names a variable in which the current entry in the vector is stored for use within the format. Note that the loop variable contains the specific database key for the vector entry and not the actual contents in the database.

If the format is specified in-line (as a double-quoted string), the loop variable, if used in the format, needs to have the dollar-sign escaped. This will prevent it from being substituted at the time the statement is being parsed and instead it will be substituted when the loop is being evaluated.

Examples:

```
@each($system.interface[], $iface)
  interface $iface->name
@if($iface->description ? "  description $iface->description\n")
@end
```

```
@each($ipv4.nameserver[], $ns, "set name-server \$ns->address\n")
```

7.4 Include statement

The *@include* statement allows the format to be split into multiple files. Using *@include* will evaluate and insert the resulting output of the included file at the place of the statement.

@include only takes one argument which is a double quoted string specifying the name of the file to include.

Example:

```
@include("sub-format.d2t")
```

8 Example

The two format files below will generate an output that looks like:

```
#
# Configuration generated at Tue Sep 3 10:12:02 CET 2013
#
set hostname Router
#
interface eth0 {
 set description LAN interface
 set ip address 1.2.3.4/24
 set ip address 5.6.7.8/28
}
interface eth1 {
 set description WAN
 set ip address 3.3.3.3/30
}
```

config.d2t

```
#
# An example db2txt format
#
@set($timestamp, $master::timestamp())
\#
\# Configuration generated at $timestamp
\#
set hostname $system.hostname->name
\#
@each($system.interface[], $if)
```

```
  interface $if->name {
  @include("interface.d2t")
  }
  @end
```

interface.d2t

```
  @if($if->description ? " description $if->description\n")
  @each($if.ipv4.addr[], $addr, " ip address $addr->address\n")
```

9 Limitations

The if-statement does not support multiple conditions using the *AND* and *OR* operators, *&&* and ||, commonly used in scripting languages. Multiple conditions must be written as multiple statements. For example, the following code is invalid.

```
@if($my.key->var1 == (int)1 && $my.key->var2 == nil)
some format...
@end
```

It must be written as follows.

```
@if($my.key->var1 == (int)1)
@if($my.key->var2 == nil)
some format...
@end
@end
```