Introduction

Active CLI (ACLI) is an SSH, Telnet and Serial Port terminal with advanced features for interacting with Extreme Networking products.

The main features are:

- Grep capability on output of any command; works properly with context-based switch configuration; simple grep, advanced grep, negative grep and no limit on chain of grep sequences (always better than switch own grep!)
- Alias capability: define aliases for the commands you use most of the time; comes with many pre-defined in *acli.alias* file which you can edit, or you can place yours in *merge.alias*
- Run multiple terminal sessions, and tie them together with sockets; issue commands in one and they get executed in all others as well
- Output redirect to, and input source from files directly from your local file system (no more hassle of getting the files to/from the switch or the inconvenience of Stackables which have no file system)
- Set variables per device; capture port lists/ranges from output; embed same variables in any commands you issue; save variables and have them automatically reloaded when connecting to same switch
- Enhanced history of commands (no more pathetic 15 commands of ERS Stackable history); because of alias and variable support, 2 histories are held: commands typed by user & commands actually sent to switch
- Localized more paging, makes session seem more responsive on slow connections/switches; terminal obtains full output in the background while user pages through output at a slower rate
- Unlimited commands can be copy-pasted into terminal; each line is fed at every prompt; terminal is capable of making difference between user typing and user pasting
- When pasting or sourcing, commands which prompt for confirmation (y/n) are automatically confirmed without having to remember to add a 'y' + Carriage return to CLI script

- When pasting or sourcing, if an error is encountered, sourcing will stop there; + ability to resume from where we left, as pasting/sourcing buffer can be recalled with @resume command
- Ability to repeat a command at configurable regular intervals, indefinitely or until user tries to interact again with CLI session
- Ability to repeat a command and replace fields within it at each iteration with list or range of values
- Ability to issue multiple commands on same line separated by semi-colon ';', which allows above functions as well as alias to be used to send multiple commands to the switch
- Automatically unwraps annoying wrapped long lines from Stackable/ISW show running-config and log file
- Suppresses annoying escape characters from stackable login; so that when capturing to log file, the file is readable afterwards (Stackable banner is reformatted and maintained)
- Ability to maintain a cache of past terminal server connections for ease of recall by mapping IP & TCP port to Switch name, model & MAC address
- The same grep capability can be leveraged on offline config files by invoking the acli command with -g switch to a single file or file wildcard or piping to *acli -g*
- For SSH publickey authentication, ability to view installed public keys on switch as well as plant own public key in the right file in the right format for the right user access
- Do not get disconnected by switch after a few minutes inactivity; ACLI terminal holds its own session timer and will generate regular keep alives to hold the session up until its own session timer expires
- Ability to highlight (e.g. in red bold) any string or pattern in output stream; handy for inspecting large log files for certain keywords
- Ability to modify and/or recolour selected output from device using sed patterns which can be defined in an offline file
- Scripting support with the ability to use control structures, error detection, user input, and controlled output
- Dictionary support, allowing online translation from loaded dictionary CLI flavour into syntax of connected device
- Ability to push(put) or pull(get) with FTP or SFTP one or many files from one or many switches simultaneously using supplied aftp

command tool
- Ability to launch many ACLI sessions from a command line using IP/hostname lists, or from a hosts file, or from a batch file; using the ACLI GUI Launcher tool
- Ability to extract device information from XMC via GraphQl API, in order to easily launch many ACLI sessions against XMC discovered devices; using the XMC ACLI Launcher tool

ACLI Terminal is written in Perl and is distributed with ConsoleZ (FreeWare) which is an improved DOS box window for use on Microsoft Windows.

The ACLI installer delivers a complete package installation for Microsoft Windows, which includes all the necessary Perl files and modules as well as ConsoleZ executable.

ACLI will also work on Linux and Apple MAC OS, but the author does not currently distribute a distribution package for these (so one would have to install all the necessary Perl modules manually).

Interactive Mode

The value of the ACLI terminal is mostly in its interactive mode of operation, which is only available with a number of Extreme devices.

To understand the ACLI interactive mode it is necessary to understand how a CLI terminal normally works.

A terminal receives user input from the keyboard. The input consists of raw characters which are simply transferred to the connected host. The terminal also displays user output to the screen (terminal window) however any character that the user inputs is not directly displayed on the screen. The terminal will only display on the screen character/text output received from the connected host. What the user types on the keyboard input is only seen on the screen output because the connected host echoes back every character which is sent to it. This is true whether the connection is run over Telnet, SSH or Serial port.

When ACLI connects to a device, it attempts to discover whether the host device is a supported Extreme Networks device. If so the terminal does a brief device discovery and enters interactive mode. (If you did not want the terminal to automatically attempt auto-discovery use the *-n* ACLI command line switch).

The auto-discovery is necessary to detect the device's base MAC address and available ports as well as some other device attributes; among these is the device family type, CLI prompt and more prompt. Some of these settings are displayed on the terminal window during device discovery:

```
EXTREME NETWORKS VOSS COMMAND LINE INTERFACE

Login: rwa
Password: ***

acli.pl: Detected an Extreme Networks device -> using terminal interactive mode
VSP-8284XSQ:1>% enable
acli.pl: Detecting device ...
acli.pl: Detected VSP-8284-XSQ (00-51-00-ca-e0-00) Single CPU system, 2 slots 84 ports
acli.pl: Use '^T' to toggle between interactive & transparent modes

VSP-8284XSQ:1#%
```

Other settings can be viewed under the ACLI control interface using the *'terminal info'* command:

```
ACLI> terminal info
acli.pl operational settings:
        AutoDetect Host Type   : enable
        Terminal Mode          : interact
        Host Capability Mode   : interact
        Host Type              : PassportERS
        Host Model             : VSP-8284-XSQ
        ACLI(NNCLI)            : yes
        Prompt Match           : 'VSP-8284XSQ:[12356](?:\((.+?)\))?[>#]'
        More Prompt Match      : '\n\x0d?--More-- \(q = quit\) |--More--'
        Suffix Prompt %        : enable
        Toggle CTRL character  : ^T
        Config indentation     : 3 space characters
        Host Error detection   : enable
        Host Error level       : error
        Keep Alive Timer       : 4 minutes
        Session Timeout        : 10 hours
        Connection Timeout     : 25 seconds
        Login Timeout          : 30 seconds
        Interact Timeout       : 15 seconds
        Newline sequence       : Carriage Return (CR)
        Negotiate Terminal Type : vt100
        Negotiate Window Size  : 132 x 24 (width/height)
```

Once in interactive mode the device's prompt is displayed with an appended '%' character and the ACLI terminal now operates as a command based terminal instead of a traditional character based terminal. In this mode, once the terminal has locked on a valid device prompt, any character typed by the user is not sent to the host, but is instead displayed on the terminal output/screen. Only once the user hits the enter key, the entered command is examined and then a decision is made as to what command is actually sent to the connected host. This is essentially the linchpin of many of the ACLI features and capabilities. For example, if the user specified some grep patterns on the submitted command, these are recorded but removed from the command before it is sent to the host (when the device output comes back, it can be processed to filter out only the output that matches the grep string); if any variables were embedded in the command, these can be replaced with the values they hold, before sending the command to the connected host; if the command matches a defined alias, then the appropriate command becomes whatever the alias refers to; and if the command is an embedded command (begins with '@') then the command is processed locally and nothing is sent to the connected host.

Hitting the return key is not the only time that the ACLI terminal will send/interact with the connected device. Hitting the TAB key or the '?' key will also trigger the same on the underlying device; this integration is fairly complex as ACLI needs to do tab expansion or syntax checking against the connected device and, depending on output, update its local command buffers in a seamless way.

To come out of interactive mode, simply hit CTRL-T which will toggle between interactive and transparent modes. The control sequence for toggling between interactive and transparent modes can be set either in the ACLI control interface or can be set in the *acli.ini* file (see ACLI ini file section).

```
ACLI> ctrl info
CTRL characters:
        Escape character     : ^]
        Quit character       : ^Q
        Terminal mode toggle : ^T
        More paging toggle   : ^P
        Send Break           : ^S
        Debug                : ^[
ACLI>
```

There are a number of CTRL keys defined within ACLI interactive mode to help with command editing. These are inspired from what is typically available on connected devices, but are purely handled on the ACLI interactive terminal side:

- **CTRL-H** or **Backspace**: Deletes one character from the right; cursor moves one space towards the left.
- **CTRL-D** or **Delete**: Deletes one character from the left; cursor does not move, any remaining characters on the right are moved one space towards the left.
- **CTRL-B** or **Cursor-Left**: Moves the cursor one character to the left.
- **CTRL-F** or **Cursor-Right**: Moves the cursor one character to the right.
- **CTRL-P** or **Cursor-Up**: Recalls previous command from history of entered commands.
- **CTRL-N** or **Cursor-Down**: Recalls next command from history of entered commands.
- **CTRL-A**: Moves the cursor to the beginning of the line.
- **CTRL-E**: Moves the cursor to the end of the line.
- **CTRL-C** or **CTRL-U**: Deletes the line; clears the current prompt for a new command.
- **CTRL-K** or **CTRL-R**: Redisplays the current command line on the current prompt.
- **CTRL-W**: Deletes word left of cursor.
- **CTRL-X**: Deletes all characters left of cursor.

Note that ConsoleZ can also reserve CTRL key sequences for its own use. If any of the above CTRL sequences is processed by ConsoleZ, then it will no longer be available to ACLI.

Interactive mode is currently supported on the following devices:

- Extreme Networks:
  - VOSS: VSP Series switches

- BOSS: ERS Series switches
- XOS: Summit Series switches
- SLX: Data Center switches
- ISW industrial switches
- Series200: models 210, 220
- Wing: APs and Controllers
- Legacy Avaya:
  - DSG white-label switches
  - ERS 8x00 Chassis
  - WLAN9100 Series
- Legacy Nortel:
  - Passport 8000 Chassis
  - Metro-ERS 8000 Chassis
  - Secure Router Series
  - WLAN8100 Series
  - WLAN2300 Controllers
  - Baystack Series switches

Transparent Mode

When in transparent mode, the ACLI terminal behaves exactly like any other SSH/Telnet/Serial port terminal and behaves like a character oriented terminal. Any characters entered by user on the keyboard are transparently sent to the connected device. And any characters/text received from the connected device is printed in the terminal output window.

By default ACLI will try and auto-detect the device during connection; if a supported Extreme Networks device is detected then ACLI will enter interactive mode, if not it will enter transparent mode. (If you did not want the terminal to automatically attempt auto-discovery use the *-n* ACLI command line switch)

Most of ACLI's advanced features will not work in transparent mode, with a few exceptions:

- If username/password credentials are provided, login is automatically performed (for any of Telnet/SSH/Serial port)
- ACLI terminals can still be tied together via sockets (to drive many terminals from one); but the socket commands to tie/listen need to be accessed via the ACLI control interface (hit CTRL-] to access)

ACLI Control Interface

Much like Telnet & FTP, the ACLI terminal offers a control interface where new connections can be initiated (using the *'open'* command) and parameters for the terminal and connection can be viewed or managed. The ACLI Control interface is what you get when you launch the ACLI named tab, which produces a prompt: *"ACLI>"*

```
Loading alias file: C:\Users\lstevens\Scripts\acli\acli.alias
Merging alias file: C:\Users\lstevens\.acli\merge.alias

ACLI>
```

The control interface can also be obtained any time, even if already connected, simply by hitting CTRL-]. If in interactive mode, it can also be obtained by invoking the *@acli* embedded command
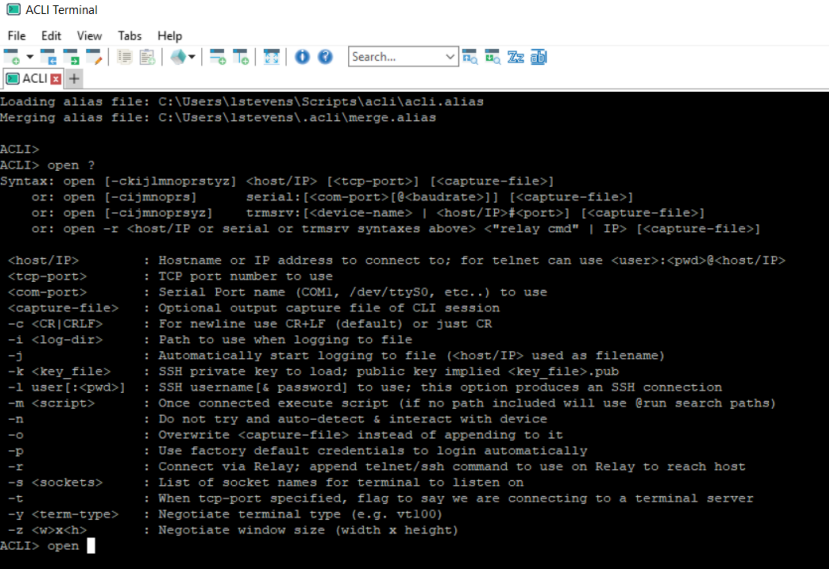
```
VSP-8284XSQ:1#% @acli
ACLI>
```

If accessing the ACLI control interface while a connection exists, then to return to the connection session simply hit enter at the *ACLI>* prompt

If running in interactive mode, most of the commands available under ACLI control interface are also available as embedded commands, so it is seldom necessary to access it.

Opening a connection

To start a connection, simply run an ACLI tab to get the ACLI control interface. Then use the *'open'* command.

```
ACLI Terminal

File  Edit  View  Tabs  Help

ACLI

Loading alias file: C:\Users\lstevens\Scripts\acli\acli.alias
Merging alias file: C:\Users\lstevens\.acli\merge.alias

ACLI>
ACLI> open ?
Syntax: open [-ckijlmnoprstyz] <host/IP> [<tcp-port>] [<capture-file>]
    or: open [-cijmnoprs]      serial:[<com-port>[@<baudrate>]] [<capture-file>]
    or: open [-cijmnoprsyz]    trmsrv:[<device-name> | <host/IP>#<port>] [<capture-file>]
    or: open -r <host/IP or serial or trmsrv syntaxes above> <"relay cmd" | IP> [<capture-file>]

<host/IP>        : Hostname or IP address to connect to; for telnet can use <user>:<pwd>@<host/IP>
<tcp-port>       : TCP port number to use
<com-port>       : Serial Port name (COM1, /dev/ttyS0, etc..) to use
<capture-file>   : Optional output capture file of CLI session
-c <CR|CRLF>     : For newline use CR+LF (default) or just CR
-i <log-dir>     : Path to use when logging to file
-j               : Automatically start logging to file (<host/IP> used as filename)
-k <key_file>    : SSH private key to load; public key implied <key_file>.pub
-l user[:<pwd>]  : SSH username[& password] to use; this option produces an SSH connection
-m <script>      : Once connected execute script (if no path included will use @run search paths)
-n               : Do not try and auto-detect & interact with device
-o               : Overwrite <capture-file> instead of appending to it
-p               : Use factory default credentials to login automatically
-r               : Connect via Relay; append telnet/ssh command to use on Relay to reach host
-s <sockets>     : List of socket names for terminal to listen on
-t               : When tcp-port specified, flag to say we are connecting to a terminal server
-y <term-type>   : Negotiate terminal type (e.g. vt100)
-z <w>x<h>       : Negotiate window size (width x height)
ACLI> open
```

To start a telnet connection:

```
ACLI> open 192.168.56.71

Logging to file: C:\Users\lstevens\Documents\ACLI-logs\192.168.56.71.log
Escape character is '^]'.
Trying 192.168.56.71
Connected to 192.168.56.71 via TELNET
acli.pl: Performing login ................
Using security software from Mocana Corporation. Please visit https://www.mocana.com/ for

Copyright(c) 2010-2018 Extreme Networks.
All Rights Reserved.
VSP Simulator: Virtual Services Platform  8200
VSP Operating System Software Build 7.1.0.0_B030 (PRIVATE)
Built: Fri Jun 29 09:09:06 EDT 2018

Unsupported Software, Internal Use Only

This product is protected by one or more US patents listed at http://www.extremenetworks.

EXTREME NETWORKS VOSS COMMAND LINE INTERFACE

Login:
```

Alternatively use the *'telnet'* command alias.

To start an SSH connection:

```
ACLI> open -l rwa 192.168.56.71

Logging to file: C:\Users\lstevens\Documents\ACLI-logs\192.168.56.71.log
Escape character is '^]'.
Trying 192.168.56.71 ..
acli.pl: Added SSH host key to known_hosts file

Enter Password:
```

```
                    .
          Connected to 192.168.56.71 via SSH
          acli.pl: Performing login ............
          Using security software from Mocana Corporation. Please visit https://www.mocana.com/ for

          Copyright(c) 2010-2018 Extreme Networks.
          All Rights Reserved.
          VSP Simulator: Virtual Services Platform  8200
          VSP Operating System Software Build 7.1.0.0_B030 (PRIVATE)
          Built: Fri Jun 29 09:09:06 EDT 2018

          Unsupported Software, Internal Use Only

          This product is protected by one or more US patents listed at http://www.extremenetworks.

          EXTREME NETWORKS VOSS COMMAND LINE INTERFACE


          acli.pl: Detected an Extreme Networks device -> using terminal interactive mode
          VSP-8284XSQ:1>% enable
          acli.pl: Detecting device ...
          acli.pl: Detected VSP-8284-XSQ (00-51-00-27-38-00) Single CPU system, 2 slots 84 ports
          acli.pl: Use '^T' to toggle between interactive & transparent modes

          VSP-8284XSQ:1#%
```

Alternatively use the *'ssh connect'* command alias.

Note, any SSH banners set on the connected device will not be displayed as it is not possible to retrieve these via the underlying Net::SSH2 Perl module used by ACLI.

And to start a serial port connection:

```
          ACLI> open -n serial:

          Known serial ports:

          Num  Serial Port         Description
          ---  -----------         -----------
            1  COM4                Standard Serial over Bluetooth link
            2  COM5                Standard Serial over Bluetooth link
            3  COM6                Prolific USB-to-Serial Comm Port

          Select entry number / serial port name glob / <entry>@<baudrate> :
```

And then select the COM port to use. Or alternatively, if you know the right COM port to use from the start:

```
          ACLI> open -n serial:COM6

          Logging to file: C:\Users\lstevens\Documents\ACLI-logs\serial_COM6.log
          Escape character is '^]'.
          Trying serial:COM6
          Connected to serial:COM6 via COM6
```

By default the baudrate is always set to 9600. To use a different baudrate, either specify it on the command line:

```
          ACLI> open -n serial:COM6@115200

          Logging to file: C:\Users\lstevens\Documents\ACLI-logs\serial_COM6.log
          Escape character is '^]'.
          Trying serial:COM6
          Connected to serial:COM6 via COM6 at baudrate 115200
```

Or enter the ACLI control interface (by hitting CTRL-]) and then set the serial port parameters using the *serial* command:

```
          ACLI> serial ?
          Syntax: serial baudrate|databits|handshake|info|parity|stopbits
```

```
ACLI> serial baudrate ?
Syntax: serial baudrate 110|300|600|1200|2400|4800|9600|14400|19200|38400|57600|115200|23
```

Note that when connecting over the serial port it is best to always use the *-n* command line switch to disable ACLI's auto-detection, which can otherwise take a while to complete over a slow 9600 baud connection (and in some cases it might fail anyway); to enter ACLI interactive mode, it is safer to hit CTRL-T once the login via serial port is complete and the switch CLI prompt has been gained.

Even easier, ACLI pre-defines a ConsoleZ tab named *'ACLI Serial Port'*:
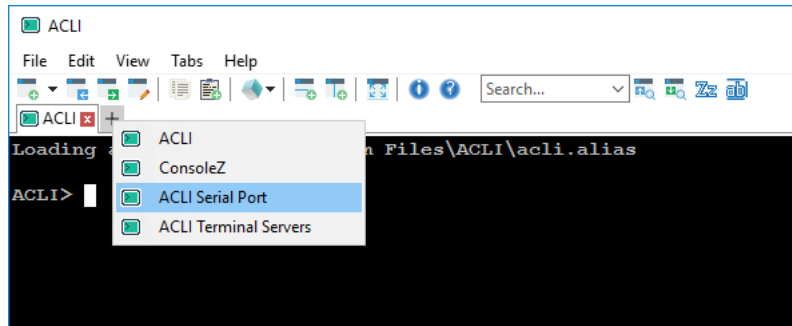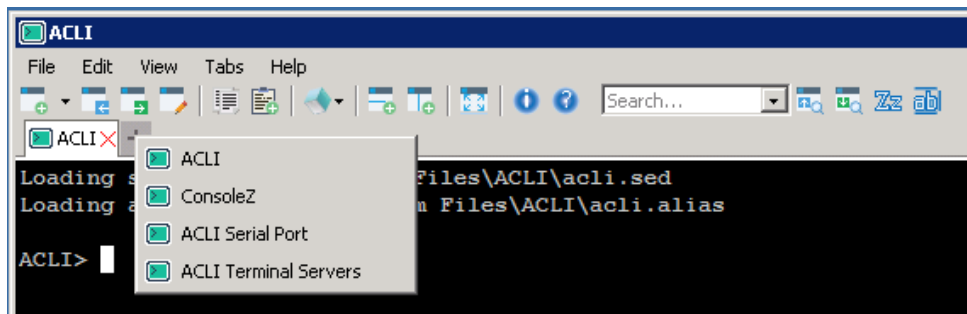

Which will execute the same:
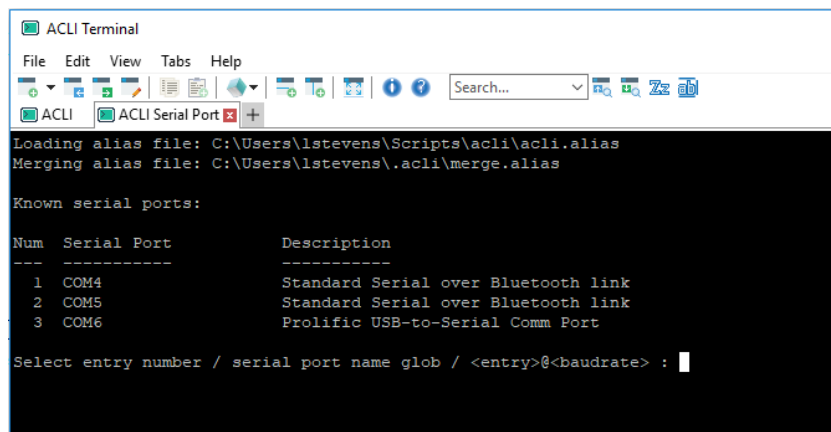
Customizing ACLI/ConsoleZ

The ACLI terminal is a Perl program (acli.pl). The graphical windows application used by ACLI is ConsoleZ which is a freeware improved console window for Microsoft Windows. (Essentially ACLI uses ConsoleZ to invoke "perl.exe acli.pl", instead of cmd.exe)

There are 4 predefined Tabs which can be launched:

- **ACLI**: this launches perl.exe onto acli.pl
- **ConsoleZ**: this launches a regular DOS box (cmd.exe)
- **ACLI Serial Port**: this launches a list of available serial ports to open ACLI against
- **ACLI Terminal Servers**: this launches a list of known terminal server connections to open ACLI against

The ConsoleZ windows can be customized to add or modify tabs. This is done under Console menu Edit / Settings / Tabs:
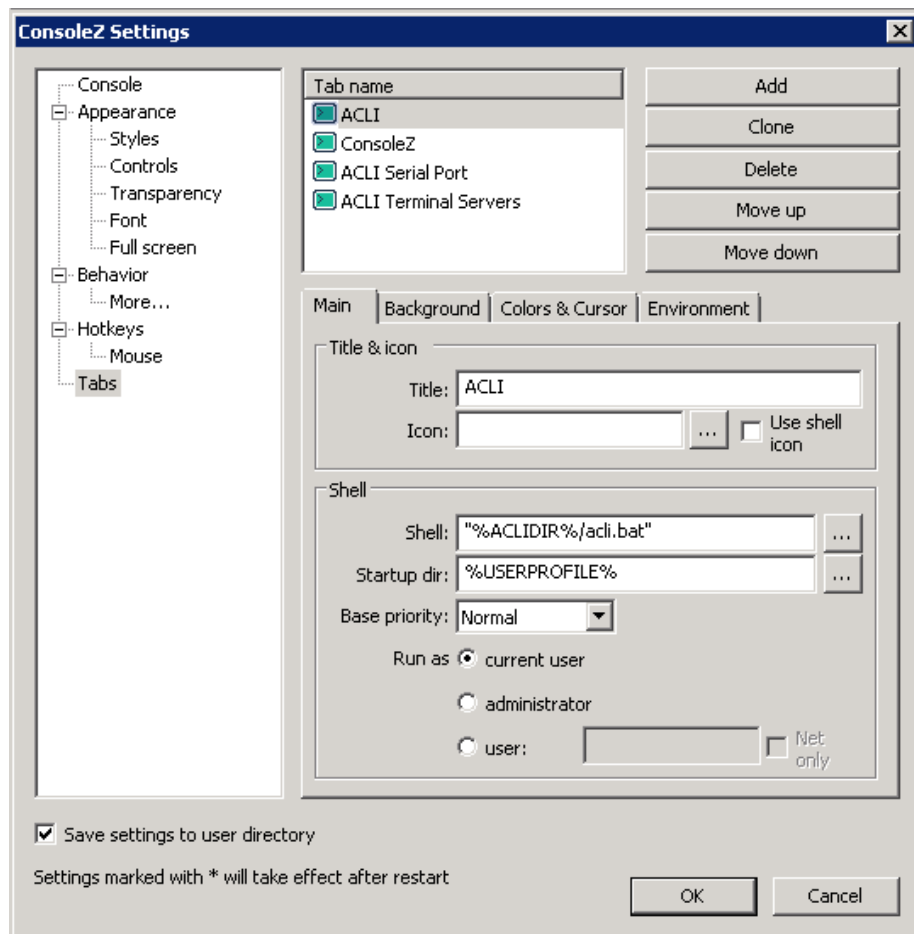
For example, new tabs can be added to connect directly to a particular switch IP, by entering these values in a new tab definition:

- **Title:** *My Switch*
- **Shell:** *"%ACLIDIR%\acli.bat" -l admin[:password] 192.168.10.1*

The above will start ACLI terminal in a new tab and will automatically connect and login to the switch IP provided. All the valid ACLI command line options can be used.

It is also possible to launch other command line applications; for example PowerShell:

- **Title:** *PowerShell*
- **Shell:** *powershell.exe*

NOTE: It is recommended that under the Console menu Edit / Settings (shown above), you check the box "Save settings to user directory"; the ConsoleZ profile (which includes your tab configurations as well as any other ConsoleZ settings) will then be saved here: *C:\Users\<username>\AppData\Roaming\Console\console.xml*. If you do not do this, the ConsoleZ profile will be saved in the ACLI install directory, with these consequences:

- If you installed ACLI "for me only": everything will work fine; but next time you install a new version of ACLI you will lose your profile settings
- If you installed ACLI "for all users" and you have Admin rights: the profile will be saved and used by all users
- If you installed ACLI "for all users" and you do not have Admin rights: you will not be able to make changes to the profile settings

Finally, you can also make ConsoleZ automatically start with selected Tabs open (and hence automatically connect to those switches) by adding the Tab names in the Windows shortcut target using the ConsoleZ -t switch; for example:

```
"C:\Program Files\ACLI\Console.exe" -w "ACLI" -t ACLI -t "My Switch"
```

This will start ConsoleZ with both the ACLI and the "My Switch" tabs open.

Embedded Commands

Embedded commands are only available in interactive mode and they always begin with the '@' character. These commands trigger some local action on the ACLI terminal and do not usually involve any communication with the connected device. Embedded commands also provide conditional statements and loop constructs for writing ACLI scripts. To view a full list of available embedded commands, simply execute the *@help* (or simply *@?*) embedded command from an interactive mode ACLI session.

```
VSP-8284XSQ:1#% @help

Embedded Commands available in interactive mode (%):

@acli                                             enter ACLI control
@alias disable|echo|enable|info|list|reload|show  show current connection aliases
@cat (or @type) <filename>                        display contents of file
@cd <relative or new directory>                   change directory
@cls or @clear                                    clear the screen
@dictionary echo|info|list|load|path|port-range|reload|unload
                                                  manage loaded dictionary
@dir (or @ls)                                     print directory
@echo on|sent|off [output on|off]|info            turns on or off displaying commands while sou
@error disable|enable|info|level                  set host error detection mode
@help or @?                                        this output
@highlight background|bright|disable|foreground|info|reverse|underline
                                                  text formatting for ^<pattern> highlights
@history [clear|device-sent|echo|info|user-entered] view or clear history
@launch                                           spawn a new ACLI session
@log auto-log|info|path|start|stop                enable/disable session logging
@ls (or @dir)                                     print directory
@mkdir <new directory to create>                  create a directory
@more disable|enable|info|lines                   enable/disable more paging
@peercp [connect|disconnect]                      view & manage peer CPU connection
@ping <hostname|ip>                               embedded ping from ACLI terminal
@print [<text>]                                   print some text; useful when sourcing and @ec
@printf "<formatting>", <value1>[,<value2>..]     print text/values with formatting (same synta
@pseudo attribute|echo|info|list|load|name|port-range|prompt|type
                                                  pseudo terminal settings
@put [<text>]                                     print some text; unlike @print, has no traili
@pwd                                              print working directory
@quit                                             quit terminal
@rediscover                                       force a full rediscovery of device
@resume [buffer]                                  resume previously interrputed sourcing or vie
@rmdir <directory to delete>                      delete a directory
@run <runscript> [$1, $2, ...args]                run script runscript.run from ACLI install or
@run list|path                                    list available run scripts, or view run scrip
@save all|delete|info|reload|sockets|vars|workdir save device variables and data
@sed colour|info|input|output|reload|reset        stream editor of input/output to/from device
@send brk|char|ctrl|string                        send special character or raw string to host
@sleep <time-in-seconds>                          pause for specified number of seconds
@socket allow|bind|disable|echo|enable|info|ip|listen|names|ping|send|tie|untie|username
                                                  link this terminal instance to others
@source <filename> [$1, $2, ...args]              source commands from file
@source <.ext> [$1, $2, ...args]                  source commands from file switchname.ext
@ssh device-keys|info|keys|known-hosts            manage SSH keys on terminal and connected swi
@status                                           show connection information
@terminal hidetimestamp|info|portrange            set selected terminal settings
@timestamp                                        print out local client's date and time
@type (or @cat) <filename>                         display contents of file
@vars [attribute|clear|echo|info|prompt|raw|show] display, clear or prompt for variables
@$ [raw|show]                                     display stored variables

Embedded Commands available only in sourced scripts:

@if <cond>, @elsif <cond>, @else, @endif          if / elsif / else conditional operators
@while <cond>, @endloop                           while loop construct
@loop, @until <cond>                              loop until construct
```

```
@my <$variable> [= <init value>]                        declare a variable which will be available on
@my <$variable1> [, <$variable2> ...]                   declare multiple variables only available in
@my <$pre_*>                                            declare variable name mask of variables only
@for <$var> &<start>..<end>[:<step>], @endfor           for loop construct using range input
@for <$var> &[']<comma-separated-list>, @endfor         for loop construct using list input (set ' to
@next [if <cond>]                                       jump to next value in a for loop construct
@last [if <cond>]                                       break out of a while, until or for loop const
@exit [if <cond>]                                       break out of sourced script
@stop ["stop-message"]                                  break out of sourced script and halts sourcin

[...]
```

Localized More Paging

When a device generates CLI output over many lines, usually the output is paged with *--more--* prompts. These allow the user to hit Space to view subsequent pages.

The ACLI terminal, in interactive mode, will also page output using more paging, but the paging is localized on the terminal and not on the connected device. What this means is that once a command has been sent to the connected device, any output produced is collected in its entirety by ACLI, but is then paged locally by ACLI on the terminal window.

This provides for a number of benefits:

- Provides a consistent output paging approach, across all supported Extreme Networks devices
- Output can be paged (by hitting Space), scrolled one line at a time (by hitting Return) and disabled on the fly (by hitting CTRL-P)
- Some ACLI features (like grep) will result in not all device produced output being displayed; consequently if the remaining output still needs to be paged, then only localized more paging can do the job properly
- When the user pages through the output, the output is displayed much faster; this is because if the user pauses between each page, in the meantime ACLI will have already obtained and cached internally the whole (or subsequent) output and so the next page can be displayed very quickly. This is particularly noticeable when there is high latency on the connection or the command is slow to produce output.
- The connected device is not held midway a CLI command if the user does not complete (or quit) the more paging. In the case of VOSS, only one user at a time can execute "show running-config"; when multiple users are connected to the same VOSS device, it is not unusual for one user to page through the config, maybe slowly, which then prevents other users from also viewing the config. This cannot happen with

ACLI, as the output of the "show running-config" is collected in one shot from the device (then locally paged to the user)

There are a few ways in which the ACLI terminal collects all output from the device, in the background, without the user even realizing. On some Extreme Networks devices, which have a fast CPU, ACLI simply disables more paging on the device; on other older Extreme devices, or connections over the serial console port (which is typically slow @ 9600 baud), ACLI keeps more paging enabled on the device, and simply keeps feeding a space character in the background (without user knowing) to retrieve all pages of output. The distinction between devices with fast or slow responsiveness is made because the user might decide to simply quit the local more paging after the first or initial pages. On slow devices, if more paging is disabled, some long commands can take many seconds to complete. In this case it is preferable to keep paging enabled on the device, so that if the user decides to quit local more paging, then ACLI can do the same on the device's own output paging, and we obtain a more responsive interaction. On more recent/faster Extreme devices, typically the whole output can be obtained from the device (with more paging disabled) even before the user has time to hit space on the first page of local more paging. Should the user decide to quit local more paging, then any output cached from the devices is simply flushed. Which behaviour is used with which Extreme device is determined during the auto-detection before entering interactive mode.

- **More paging ENABLED on device**: slow connection or device with slow CPU
    - ACLI connection over serial port
    - ACLI connection over Terminal Server
    - ACLI Telnet/SSH connection to BaystackERS older models ERS4500, ERS2500, ES470, BPS-2000, Baystack450..
    - ACLI Telnet/SSH connection to SecureRouter, WLAN2300, Accelar
- **More paging DISABLED on device**: fast connection + device with fast CPU
    - ACLI Telnet/SSH connection to BaystackERS recent models ERS3500/3600/4800/4900/5900, VSP7024

- ACLI Telnet/SSH connection to PassportERS, VOSS, ExtremeXOS, ISW, WLAN9100

A further operation mode is sync mode; in sync mode the device more paging mode is kept in synch with the ACLI more paging mode. The ACLI more paging mode can be set or viewed via the embedded *@more* command, or via the *'more'* command under ACLI control interface, or can be toggled on/off using CTRL-P.

```
VSP-8284XSQ:1#% @more ?
Syntax: @more disable|enable|info|lines|sync

VSP-8284XSQ:1#% @more info

Local more paging                              : enabled
Lines per page                                 : 22
Toggle CTRL character                          : ^P
Local paging mode synchronized on device : disabled
Underlying device more paging mode       : disabled

VSP-8284XSQ:1#%
```

In sync mode, when user disables more paging on ACLI, then more paging is also disabled on the connected device; whereas if more paging is enabled on ACLI then it is also enabled on the connected device. Sync mode is useful when dumping really large tables (e.g. FDB or ARP cache in scaled environments, with thousands of entries, and it is desired to simply quit the output at the first page, simply to inspect the summary of the number of entries present.

Sync Mode	ACLI More Paging	More Paging on device (fast CPU)	More Paging on device (slow CPU)
Disabled	Enabled	Disabled	Enabled
Disabled	Disabled	Disabled	Enabled
Enabled	Enabled	Enabled	Enabled

| Enabled | Disabled | Disabled | Disabled |

Command History

The ACLI terminal, in interactive mode, operates in a command oriented fashion. In this mode the actual command sent to the connected device can be different from the command that the user actually typed in. For example the command might include a grep string (which is removed and not sent to the device), or a variable (which is de-referenced before sending), or an alias.

ACLI actually maintains four separate CLI histories:

- A user-entered command history; this is a history of all commands actually entered by the user. This history can be viewed with the following command:

  ```
  @history user-entered

  ACLI> history user-entered
  ```

- A device-sent command history; this is a history of all commands as sent to the connected device. This history matches the history maintained by the connected device, if supported. This history can be viewed with the following command:

  ```
  @history device-sent

  ACLI> history device-sent
  ```

- A no-error-device command history; this is a history of all commands as sent to the connected device which did not generate an error from the device. This history variant is useful when using the ACLI dictionary functionality. This history can be viewed with the following command:

  ```
  @history no-error-device

  ACLI> history no-error-device
  ```

- A recall command history; a distilled list of user-entered command history, where no command is duplicate and the order is always rearranged to give the most recent command first. These commands can be recalled simply using the cursor keys or via *!<n>*. This history can be viewed with the following command:

  ```
  @history

  ACLI> history recall
  ```

All of the above histories can be inspected or cleared using the *@history* embedded command (in interactive mode) or the *'history'* command under the ACLI> control interface.

```
@history [clear|device-sent|echo|info|no-error-device|user-entered]

ACLI> history clear|device-sent|echo|info|no-error-device|recall|user-entered
```

ACLI does not limit the size of these histories (unlike the devices which typically limit their history size to some maximum number of commands).

Another nice property of ACLI history is that, since these are held by the terminal itself, they are preserved across device reconnection, or device reboot

```
VSP-8284XSQ:1#% show user
SESSION   USER                                              ACCESS   IP A
Telnet0   rwa                                               rwa      192.
Telnet1   rwa                                               rwa      192.
Console                                                     none     ----
VSP-8284XSQ:1#%
VSP-8284XSQ:1#% !!
        history% show user
SESSION   USER                                              ACCESS   IP A
```

```
Telnet0   rwa                                          rwa     192.
Telnet1   rwa                                          rwa     192.
Console                                                none    ----
VSP-8284XSQ:1#%
```

The example above uses double bang '!!' to recall the last command entered. The example below inspects the recall history and then a command is recalled by index.

```
VSP-8284XSQ:1#% @history

    1 : show snmp-server user
    2 : mlt
    3 : show snmp-server context
    4 : show snmp-server community
    5 : show snmp-server view
    6 : ipr
    7 : ipa
    8 : ipr 512
    9 : vlni
   10 : show vlan members
   11 : igi
   12 : @history
   13 : show user
   14 : show user

VSP-8284XSQ:1#% !1
        history% show snmp-server user
Engine ID = 80:00:08:E0:03:00:51:00:CA:E0:00


=============================================================================================
                                USM Configuration
=============================================================================================
User/Security Name    Engine Id                       Protocol
---------------------------------------------------------------------------------------------
initial               0x80:00:08:E0:03:00:51:00:CA:E0:00 NO AUTH,  NO  PRIVACY

2 out of 1 Total entries displayed
--------------------------------------------------------------------------------
VSP-8284XSQ:1#%
```

Notice that when a history recall is made, ACLI will add an echo line immediately after indicating the full command that was recalled. History echoing is by default enabled but can be disabled using the *'@history echo'* embedded command or the *'history echo'* command under ACLI control interface.

Unwrapping Lines

The Extreme Networks ERS (BaystackERS) platform CLI has a concept of screen width which is enforced on all lines of output. By default an ERS device will use a screen width of 80 characters. The ACLI terminal will always max that value out on the device to 131 (not the maximum 132 as this has other side effects). Increasing the screen width has the benefit of greatly reducing the number of lines of output which the ERS might have to wrap, as now only lines > 131 characters will be wrapped. However, even so, the ERS config files, particularly with eapol config lines, can often exceed even 131 characters; the same is true for the ERS and ISW log files.

The ACLI terminal provides for easy grep capability of all output, whether we are displaying the running-config or displaying the log file. However the output lines must not be wrapped or else the grep function will not behave as expected.

To solve this problem, ACLI is capable of detecting long lines (=131 characters on ERS, or wrapped ISW log lines) where the subsequent line is in fact a wraped portion of the same line. It is therefore capable of unwrapping these lines. However ACLI will only do this on CLI commands where the *-i* switch was provided (see section on ACLI command line switches).

The pre-defined *cfg* and *log* aliases thus always automatically set the *-i* switch.

Suppression of unprintable login banners

The Extreme Networks ERS (BaystackERS) platform CLI makes use of ANSI escape sequences to print the login spash banners. These banners are mostly a nuisance and if one is logging output to file, the resulting log file is just a big pile of gibberish.

The ACLI terminal reformats the ERS login banner into printable text only. Other ERS login pages, like confirmation of last login and login page are eliminated in favour of a login approach more similar to what is used with other devices.

Suppression of show command timestamp banners

Some Extreme switch family types produce a timestamp banner before every CLI "show" command. The intent is to preserve a timestamp if the output was being captured to file and inspected later. However these timestamps pollute the output window with even more uninteresting output (usually a 3 line banner) so the ACLI terminal has the ability to suppress these timestamp banners. To do so simply activate '*terminal hidetimestamp*' under ACLI control interface or set the '*hide_timestamps_flg*' key in *acli.ini*.

Note that if enabled, this feature will only suppress the timestamps from the ACLI output window. If the ACLI session is being captured to file, the timestamps are not suppressed in the output file. The following example illustrates the feature.

```
VSP-8284XSQ:1#% @log start mysession.log
Logging to file: C:\Users\lstevens\Local-Documents\ACLI-logs\mysession.log

VSP-8284XSQ:1#% vln
        alias% show vlan basic
********************************************************************************
              Command Execution Time: Sun Dec 22 17:03:34 2019 UTC
********************************************************************************
================================================================================
                                  Vlan Basic
================================================================================
VLAN                          MSTP
ID    NAME            TYPE    INST_ID PROTOCOLID  SUBNETADDR    SUBNETMASK
--------------------------------------------------------------------------------
1     Default         byPort  0       none        N/A           N/A

All 1 out of 1 Total Num of Vlans displayed
acli.pl: Displayed Record Count = 1

VSP-8284XSQ:1#%
ACLI> terminal hidetimestamp enable
ACLI>
VSP-8284XSQ:1#% vln
        alias% show vlan basic
================================================================================
                                  Vlan Basic
================================================================================
VLAN                          MSTP
ID    NAME            TYPE    INST_ID PROTOCOLID  SUBNETADDR    SUBNETMASK
--------------------------------------------------------------------------------
1     Default         byPort  0       none        N/A           N/A

All 1 out of 1 Total Num of Vlans displayed
acli.pl: Displayed Record Count = 1

VSP-8284XSQ:1#% @log stop
```

Notice that the timestamp banner is absent after enabling the feature from the ACLI control interface.

Inspection of the captured file will show the timestamp banner preserved in both instances:

```
=~=~=~=~=~=~=~=~=~=~=~= acli.pl log Sun Dec 22 18:03:29 2019 =~=~=~=~=~=~=~=~=~=~=~=
VSP-8284XSQ:1#% vln
        alias% show vlan basic
********************************************************************************
              Command Execution Time: Sun Dec 22 17:03:34 2019 UTC
********************************************************************************
================================================================================
                                  Vlan Basic
================================================================================
VLAN                          MSTP
ID    NAME            TYPE    INST_ID PROTOCOLID  SUBNETADDR    SUBNETMASK
--------------------------------------------------------------------------------
```

```
1     Default        byPort      0        none       N/A          N/A

All 1 out of 1 Total Num of Vlans displayed
acli.pl: Displayed Record Count = 1

VSP-8284XSQ:1#% VSP-8284XSQ:1#% vln
         alias% show vlan basic
********************************************************************************
               Command Execution Time: Sun Dec 22 17:03:47 2019 UTC
********************************************************************************
================================================================================
                                    Vlan Basic
================================================================================
VLAN                           MSTP
ID    NAME           TYPE      INST_ID PROTOCOLID  SUBNETADDR   SUBNETMASK
--------------------------------------------------------------------------------
1     Default        byPort      0        none       N/A          N/A

All 1 out of 1 Total Num of Vlans displayed
acli.pl: Displayed Record Count = 1

VSP-8284XSQ:1#% @log stop

=~=~=~=~=~=~=~=~=~=~=~= acli.pl log Sun Dec 22 18:04:00 2019 =~=~=~=~=~=~=~=~=~=~=~=
```

Session Logging

The ACLI terminal can log CLI sessions to file. Logging to file can be specified either when first launching ACLI from command line, or when connecting from the ACLI> control interface using the *open/telnet/ssh* commands. Logging can also be started or stopped once already connected using either the embedded *@log* commands (if in interactive mode) or the *log* command in the ACLI control interface.

```
@log start <capture-file> [-o|overwrite]

ACLI> log start <capture-file> [-o|overwrite]
```

An auto-logging is also available, where ACLI can be made to automatically log to file all connections. The functionality can be enabled via the *'@log auto-log'* command, or via the ACLI control interface *'log auto-log'* command, or command line switch (*-j*) and also via the *acli.ini* file.

```
@log auto-log disable|enable|retry

ACLI> log auto-log disable|enable|retry

VSP-8284XSQ:1#% @log ?
Syntax: @log auto-log|info|path|start|stop

VSP-8284XSQ:1#% @log info

Logging path    : C:\Users\lstevens\Documents\ACLI-logs
Logging to file : C:\Users\lstevens\Documents\ACLI-logs\192.168.56.71.log
Auto-Logging    : enabled

VSP-8284XSQ:1#%
```

The filename used is the IP address or hostname used for the connection which can be pre-pended with a timestamp string. The use and format of the timestamp string can be set in *acli.ini* via the *auto_log_filename_str* setting. See the ACLI ini file section.

A logging path directory can also be set; if set, any logging will create the session log file in the provided path and not in the working directory. The logging directory can be set via the *'@log path'* embedded command, or via ACLI control interface *'log path'* command, or command line switch (*-i*) and via the *acli.ini* file using the *log_path_str* key.

```
@log path set '<directory to use for logging>'

ACLI> log path set '<directory to use for logging>'
```

Output Redirection

The ACLI terminal supports output redirection in interactive mode. For any command (even embedded commands) the resulting output can be redirected to a file on the local file system where ACLI terminal is running simply by appending the command with '>' or '>>' and the destination filename ('>' and '>>' can also be used to capture output values into variables; in this case they are followed by a '$' variable and not a filename; this is covered in the variables section).

In both cases the file will be created if it did not exist; if the file already existed, a '>' redirection will overwrite the file contents, while a '>>' will append. If a path was not specified on the filename then the file will be created in the working directory (which can be inspected with *@pwd* and set with *@cd* embedded commands)

NOTE: This output redirection is NOT to the switch/device's own filesystem, but only to the file system of the end station where the ACLI Terminal is running

The fact that redirection occurs to the local file system of ACLI, has a number of advantages. For a start, if a user wants to capture output of a command, he will most likely want to recover that output from the switch; so with ACLI there is no need to transfer the output file from the switch. Also, when driving/tie-ing many ACLI terminals together (using sockets) it is possible to capture the output of a command (e.g. the running-config) against all switches with one single command and have the output from all switches stored locally in the same directory.

```
VSP-8284XSQ:1#% @pwd

Working directory is:
C:\Users\lstevens\Scripts\acli\working-dir

VSP-8284XSQ:1#% $$

$$             = VSP-8284XSQ
```

```
VSP-8284XSQ:1#% cfg > $$.cfg
          vars% cfg > VSP-8284XSQ.cfg
         alias% show running-config -ib > VSP-8284XSQ.cfg

acli.pl: Saving output ..done
acli.pl: Output saved to:
C:\Users\lstevens\Scripts\acli\working-dir\VSP-8284XSQ.cfg

VSP-8284XSQ:1#%
```

Grep

Grep is the ability to filter output lines based on matching a string or pattern within them. Grep capability is one of the major features of the ACLI terminal. In interactive mode any CLI command can be followed by any number of grep strings and of various types. The grep strings are processed locally by the ACLI terminal and are stripped off the CLI command before it is sent to the connected host. When the host sends the output back, the ACLI terminal filters that output based on what grep patterns had been defined by the user. ACLI supports these four basic grep types:

```
<CLI command> |  <grep string> [-s]           simple line grep
<CLI command> ! [<grep string> [-s]]          simple line negative grep
<CLI command> ^ <match string> [-s]           highlight matched string in output st
<CLI command> || <grep string> [-s]           advanced grep
<CLI command> !! <grep string> [-s]           advanced negative grep
```

Where the optional -*s* switch makes the grep string provided case sensitive. Most of the time when filtering switch output with grep we care little about case sensitivity, so the ACLI terminal defaults to a case insensitive grep, and offers the optional -*s* switch for the less frequent case sensitive greps.

Simple Grep

Simple grep is about only displaying lines in which the grep string or pattern is found.

```
VSP-8284XSQ:1#% show running-config | ssh
boot config flags sshd
# SSH CONFIGURATION
ssh

VSP-8284XSQ:1#%
```

Multiple grep patterns can be provided and separated by comma; this acts as a logical OR on those patterns, if any of them is matched, then the lines are displayed.

```
VSP-8284XSQ:1#% show running-config | ssh,telnet
boot config flags sshd
boot config flags telnetd
# SSH CONFIGURATION
ssh

VSP-8284XSQ:1#%
```

You can also concatenate multiple grep strings to the same CLI command; this acts as a logical AND on those patterns, only lines matching all the grep patterns will be displayed. The ACLI grep implementation does not impose any limit on how many grep strings are concatenated.

```
VSP-8284XSQ:1#% show running-config | ssh,telnet | flag
boot config flags sshd
boot config flags telnetd

VSP-8284XSQ:1#%
```

Simple Negative Grep

Simple negative grep is about only displaying lines in which the grep string or pattern is not found. Negative grep usually makes more sense when concatenated with positive grep so as to remove some unwanted output from the output produced from the previous grep function.

```
VSP-8284XSQ:1#% show running-config | ssh,telnet ! ^#
boot config flags sshd
boot config flags telnetd
ssh

VSP-8284XSQ:1#%
```

Notice that we used the negative grep to remove the banner line beginning with '#'. Also notice that we were able to concatenate a positive grep with a negative grep in the above example. The ACLI grep implementation allows any grep type (simple/advanced, positive/negative) to be concatenated as many times as desired.

A special case of negative grep is when it used without any grep string(i.e. just a trailing '!'). In this case it will remove all empty lines from the output. This form of negative grep can only be combined with other greps if it is the rightmost.

```
VSP-8284XSQ:1#% show qos ingressmap 1p
================================================================================
                    Qos Ingress IEEE 1P to QOS-Level Map
================================================================================
IEEE1P          QOSLEVEL
--------------------------------------------------------------------------------
0               1

1               0

2               2

3               3

4               4

5               5

6               6

7               7

VSP-8284XSQ:1#% show qos ingressmap 1p !
================================================================================
                    Qos Ingress IEEE 1P to QOS-Level Map
================================================================================
IEEE1P          QOSLEVEL
--------------------------------------------------------------------------------
0               1
1               0
2               2
3               3
4               4
5               5
6               6
7               7

VSP-8284XSQ:1#%
```

Highlight

The highlight function with '^' is already covered in a separate section. It is also included here since it is extremely similar to the simple and negative greps in the way it works and behaves in exactly the same way with regards to all of the considerations covered in the next section. The only difference between highlight and simple greps is that the highlight will not filter out any lines and instead will simply apply the highlight where the grep string (or match string in this case) matches. The highlight function can also be concatenated with grep functions on the only condition that there can be only one highlight function and it needs to be the very last in the concatenation.

Considerations around ACLI grep

Note that spaces before/after the grep pipe '|' / bang '!' characters are optional and can be omitted. Also any spaces before or after the provided grep string/pattern are automatically removed (only spaces inside the grep string are preserved). If you need to include leading or trailing spaces in the grep pattern, simply enclose the grep pattern in quotes (single or double quotes).

```
VSP-8284XSQ:1#% show running-config|ssh,telnet!^#
boot config flags sshd
boot config flags telnetd
ssh

VSP-8284XSQ:1#%

VSP-8284XSQ:1#% show interfaces gigabitEthernet interface | up
1/1      192    10GbNone         true  false  1950  00:51:00:ca:e0:00 up      up
1/2      193    10GbNone         true  false  1950  00:51:00:ca:e0:01 up      up
1/3      194    10GbNone         true  false  1950  00:51:00:ca:e0:02 up      up
1/4      195    10GbNone         true  false  1950  00:51:00:ca:e0:03 up      down
VSP-8284XSQ:1#%
VSP-8284XSQ:1#% show interfaces gigabitEthernet interface | "  up"
1/1      192    10GbNone         true  false  1950  00:51:00:ca:e0:00 up      up
1/2      193    10GbNone         true  false  1950  00:51:00:ca:e0:01 up      up
1/3      194    10GbNone         true  false  1950  00:51:00:ca:e0:02 up      up
VSP-8284XSQ:1#%
```

The last example shows how to specify a grep pattern of *" up"* with 2 preceding spaces such that we can filter on the last column containing 'up' and not the penultimate column. Or again, we could use negative grep to achieve the same:

```
VSP-8284XSQ:1#% show interfaces gigabitEthernet interface | up ! down
1/1      192    10GbNone         true  false  1950  00:51:00:ca:e0:00 up      up
1/2      193    10GbNone         true  false  1950  00:51:00:ca:e0:01 up      up
1/3      194    10GbNone         true  false  1950  00:51:00:ca:e0:02 up      up
VSP-8284XSQ:1#%
```

ACLI's grep takes some special consideration whenever the grep string is a number or an Ethernet port number (in any of the possible formats slot/port, slot/port/channel, slot:port). It would be particularly annoying if having requested a grep of '2' we got back lines with '12', '22', etc..; likewise, if our grep string is '1/1' we are unlikely to want to see lines referring to ports 1/10, 1/11, etc... So the ACLI grep is being a bit more clever when it is given a number or a port number:

```
EXOS-VM.19 #% if | 2
        alias% show ports information | 2
2          Dm------e--fMB----- ready    - / -  1  1   1    1 9216 none
EXOS-VM.19 #%
```

The example above might not seem that obvious, so we will just show what you would get if using the switch's own pipe include grep function (i.e. the switch CLI is doing the grep, not the ACLI terminal):

```
EXOS-VM.19 #% show ports information \| include 2
1          Em------e--fMB----- active   - / -  1  1   1    1 9216 none
2          Dm------e--fMB----- ready    - / -  1  1   1    1 9216 none
3          Dm------e--fMB----- ready    - / -  1  1   1    1 9216 none
EXOS-VM.20 #%
```

Note that the switch CLI is hopeless and shows all 3 ports (XOS VM used here) since a '2' appears in the MTU value '9216'. Note as well that if you really wanted to use the switch CLI to perform the grep, you need to back slash the pipe '|' character (or else toggle into ACLI transparent mode with CTRL-T), otherwise ACLI will perform the grep instead.

In the unlikely case where you would actually like ACLI's grep to do a raw and stupid grep (like the switch CLI does) then all you need to do is to place quotes around the number, so ACLI will treat the grep string as text:

```
EXOS-VM.21 #% if | "2"
         alias% show ports information | "2"
1       Em------e--fMB----- active   - / -   1   1    1    1 9216 none
2       Dm------e--fMB----- ready    - / -   1   1    1    1 9216 none
3       Dm------e--fMB----- ready    - / -   1   1    1    1 9216 none
EXOS-VM.21 #%
```

We will just demonstrate the same point using port numbers in the slot/port format, using VOSS this time:

```
VSP-8284XSQ:1#% if | 1/1
         alias% show interfaces gigabitEthernet interface!!locked | 1/1
1/1     192   10GbNone           true  false   1950  00:51:00:ca:e0:00 up     up
VSP-8284XSQ:1#%
```

Only a line with port 1/1 is shown. Yet if we tried to use VOSS's own CLI grep function we get hopeless output:

```
VSP-8284XSQ:1#% show interfaces gigabitEthernet interface \| include 1/1
1/1     192   10GbNone           true  false   1950  00:51:00:ca:e0:00 up     up
1/10    201   10GbNone           true  false   1950  00:51:00:ca:e0:09 down   down
1/11    202   10GbNone           true  false   1950  00:51:00:ca:e0:0a down   down
1/12    203   10GbNone           true  false   1950  00:51:00:ca:e0:0b down   down
1/13    204   10GbNone           true  false   1950  00:51:00:ca:e0:0c down   down
1/14    205   10GbNone           true  false   1950  00:51:00:ca:e0:0d down   down
1/15    206   10GbNone           true  false   1950  00:51:00:ca:e0:0e down   down
1/16    207   10GbNone           true  false   1950  00:51:00:ca:e0:0f down   down
1/17    208   10GbNone           true  false   1950  00:51:00:ca:e0:10 down   down
1/18    209   10GbNone           true  false   1950  00:51:00:ca:e0:11 down   down
1/19    210   10GbNone           true  false   1950  00:51:00:ca:e0:12 down   down
VSP-8284XSQ:1#%
```

And if we wanted the same hopeless output with ACLI's grep, then we could simply put quotes around our port number:

```
VSP-8284XSQ:1#% if | "1/1"
         alias% show interfaces gigabitEthernet interface!!locked | "1/1"
1/1     192   10GbNone           true  false   1950  00:51:00:ca:e0:00 up     up
1/10    201   10GbNone           true  false   1950  00:51:00:ca:e0:09 down   down
1/11    202   10GbNone           true  false   1950  00:51:00:ca:e0:0a down   down
1/12    203   10GbNone           true  false   1950  00:51:00:ca:e0:0b down   down
1/13    204   10GbNone           true  false   1950  00:51:00:ca:e0:0c down   down
1/14    205   10GbNone           true  false   1950  00:51:00:ca:e0:0d down   down
1/15    206   10GbNone           true  false   1950  00:51:00:ca:e0:0e down   down
1/16    207   10GbNone           true  false   1950  00:51:00:ca:e0:0f down   down
1/17    208   10GbNone           true  false   1950  00:51:00:ca:e0:10 down   down
1/18    209   10GbNone           true  false   1950  00:51:00:ca:e0:11 down   down
1/19    210   10GbNone           true  false   1950  00:51:00:ca:e0:12 down   down
VSP-8284XSQ:1#%
```

As well as properly dealing with numbers and port numbers, ACLI's grep is also able to take numerical or port ranges as search string and to find all hits for all numbers/ports in the range.

```
EXOS-VM.21 #% if || 1-3
         alias% show ports information || 1-3
Port       Flags              Link    ELSM Link Num  Num   Num Jumbo QOS     Load
                              State    /OAM UPS STP VLAN Proto  Size profile Master
=================================================================================
1          Em------e--fMB----- active  - / -  1   1    1     1 9216 none
2          Dm------e--fMB----- ready   - / -  1   1    1     1 9216 none
3          Dm------e--fMB----- ready   - / -  1   1    1     1 9216 none
=================================================================================
EXOS-VM.21 #%
```

Note that port 2 was included in the grep output.

```
VSP-8284XSQ:1#% if | 1/1-1/5
         alias% show interfaces gigabitEthernet interface!!locked | 1/1-1/5
1/1     192   10GbNone           true  false   1950  00:51:00:ca:e0:00 up     up
1/2     193   10GbNone           true  false   1950  00:51:00:ca:e0:01 up     up
```

```
1/3      194   10GbNone         true  false    1950  00:51:00:ca:e0:02 up      up
1/4      195   10GbNone         true  false    1950  00:51:00:ca:e0:03 up      down
1/5      196   10GbNone         true  false    1950  00:51:00:ca:e0:04 down    down
VSP-8284XSQ:1#%
```

Note that ports 1/2,1/3 & 1/4 were included.

All the points covered in this section are also applicable to the advanced grep variants which will be covered below.

Use of regular expressions

In one of the previous examples, the caret sign '^' was used as the standard regular expression for matching text anchored at the beginning. ACLI grep supports the majority of Perl's regular expressions, though for some the character involved is overloaded by ACLI so might need back slashing; follows a list of the most commonly used.

- **Metacharacters**
  - **^** : Match at the beginning of the line. The '^' character has to appear as the first character after the grep pipe '|' or bang '!' grep identifier (otherwise it will be interpreted as ACLI highlight feature)
  - **$** : Match at the end of the line. The '$' character has to appear as the last character of the grep string/patterns, not followed by any other character (otherwise ACLI will think it is referencing a $variable)
  - **.** : Match any character. The '.' character always shows up in IP addresses, so if you do a grep for IP 192.168.10.1 the dots will match any character, not just a real dot; one would have to do a grep for 192\.168\.10\.1 in order to force matching a real dot, but in practice no point to make one's life too complicated... as there is probably nothing but an IP address which will match in the output anyway. Just be aware of what you are actually matching.
  - **()** : Round brackets are used for grouping, which can be used for capturing (not applicable to ACLI, except in variable capturing with regex syntax) or for associating the grouping with quantifiers or for performing alternation inside the grouping using the '|' character (next one)
  - **|** : Alternation can be used with ACLI, but only if inside grouping brackets and the pipe '|' character needs to be back slashed (otherwise it is interpreted by ACLI as a new grep pattern) like this: *(<pattern1>\|<pattern2>)*

        VSP-8284XSQ:1#% sys | 00:51:00:ca:e0:(00\|81)
               alias% show sys-info | 00:51:00:ca:e0:(00\|81)
            BaseMacAddr   : 00:51:00:ca:e0:00
            MgmtMacAddr   : 00:51:00:ca:e0:81

        VSP-8284XSQ:1#%

  - **[]** : Bracketed character class. Match one character against a range of different possible characters. For example [abc] will match a single character as long as that character is either 'a' or 'b' or 'c'; [a-zA-Z] will match any character in the alphabet either lower or upper case.

        VSP-8284XSQ:1#% sys | 00:51:00:ca:e0:[08][01]
               alias% show sys-info | 00:51:00:ca:e0:[08][01]
            BaseMacAddr   : 00:51:00:ca:e0:00
            MgmtMacAddr   : 00:51:00:ca:e0:81

        VSP-8284XSQ:1#%

- **Character Classes**
  - **\w** : Match a word character, i.e. alphanumeric plus underscore '_'
  - **\W** : Match a non-word character (opposite of above)
  - **\s** : Match a whitespace character (includes both spaces and tabs)
  - **\S** : Match a non-whitespace character (opposite of above)
  - **\t** : Match the tab character
  - **\d** : Match a decimal digit character (0-9)
  - **\D** : Match a non-digit character

- **Quantifiers**

  The quantifiers below can be applied immediately after a character or the above character classes or for a grouping enclosed in brackets '()'

  - ***** : Match 0 or more times

- + : Match 1 or more times
- **?** : Match 0 or 1 times
- **{n}** : Match exactly n times
- **{n,}** : Match at least n times
- **{n,m}** : Match at least n but not more than m times

Note that for the last three quantifiers, the '{}' characters are overloaded by ACLI (they are used to embed Perl code snippets in CLI command lines) so to use them as quantifiers they will need to be back slashed (and only the first '{' needs back slashing: **\{n}, \{n,}, \{n,m}**

```
VSP-8284XSQ:1#% sys | (.00)\{1}
         alias% show sys-info | (.00){1}
      BaseMacAddr         : 00:51:00:ca:e0:00
      MgmtMacAddr         : 00:51:00:ca:e0:81
         Last Change: 0 day(s), 00:00:41   (0 day(s), 02:37:55 ago)
    Last Vlan Change: 0 day(s), 00:00:41   (0 day(s), 02:37:55 ago)
Last Statistic Reset: 0 day(s), 00:00:00

VSP-8284XSQ:1#% sys | (.00)\{2,3}
         alias% show sys-info | (.00){2,3}
         Last Change: 0 day(s), 00:00:41   (0 day(s), 02:38:28 ago)
    Last Vlan Change: 0 day(s), 00:00:41   (0 day(s), 02:38:28 ago)
Last Statistic Reset: 0 day(s), 00:00:00

VSP-8284XSQ:1#% sys | (.00)\{3,}
         alias% show sys-info | (.00){3,}
Last Statistic Reset: 0 day(s), 00:00:00

VSP-8284XSQ:1#%
```

For a complete reference on Perl regular expressions refer to the following link: https://perldoc.perl.org/perlre.html

To note that ACLI will always process the above Perl metacharacters, provided that they are provided in a non-quoted string or in a double quoted string. If instead the string is single quoted then none of the above metacharacters will be processed. For example, if you wanted to match(/highlight) a particular VSP log file timestamp, this command would not work:

```
VSP8200-1:1#% show log file | 2020-04-23T23:00:29.894+00:00
VSP8200-1:1#%
```

That is because the '+' character is interpreted as the 1 or more times quantifier and hence ACLI will be looking for lines where 4 is repeated 1 or more times followed by a 0 (with no '+' character in between), which will never match if that particular timestamp is in the log file. Enclosing the timestamp in single quotes solves the problem:

```
VSP8200-1:1#% show log file | '2020-04-23T23:00:29.894+00:00'
1 2020-04-23T23:00:29.894+00:00 VSP8200-1 CP1 - 0x000305cf - 00000000 GlobalRouter SW INF
1 2020-04-23T23:00:29.894+00:00 VSP8200-1 CP1 - 0x000045e3 - 00000000 GlobalRouter SNMP I
VSP8200-1:1#%
```

For reference, we could have also solved the problem by backslashing the '+' character:

```
VSP8200-1:1#% show log file | 2020-04-23T23:00:29.894\+00:00
1 2020-04-23T23:00:29.894+00:00 VSP8200-1 CP1 - 0x000305cf - 00000000 GlobalRouter SW INF
1 2020-04-23T23:00:29.894+00:00 VSP8200-1 CP1 - 0x000045e3 - 00000000 GlobalRouter SNMP I
VSP8200-1:1#%
```

Advanced Grep

The simple line grep and negative greps covered so far are good for many uses, but fall short of many other use cases. For example, most CLI "show" commands start off with a handy banner which labels the various columns of output that follow:

```
VSP-8284XSQ:1#% show interfaces gigabitEthernet name
================================================================================
                                    Port Name
================================================================================
PORT                                          OPERATE  OPERATE  OPERATE
NUM      NAME                    DESCRIPTION   STATUS   DUPLX    SPEED    VLAN
--------------------------------------------------------------------------------
1/1                              10GbNone      up       full     10000    Acces
1/2                              10GbNone      up       full     10000    Acces
1/3                              10GbNone      up       full     10000    Acces
1/4                              10GbNone      down     full     0        Acces
1/5                              10GbNone      down     full     0        Acces
1/6                              10GbNone      down     full     0        Acces
```

If we apply a simple grep to such a command we end up with the lines that contain our seeked grep string but, chances are, we lose most of the banners in the process:

```
VSP-8284XSQ:1#% show interfaces gigabitEthernet name | up
NUM      NAME                    DESCRIPTION   STATUS   DUPLX    SPEED    VLAN
1/1                              10GbNone      up       full     10000    Acces
1/2                              10GbNone      up       full     10000    Acces
1/3                              10GbNone      up       full     10000    Acces
VSP-8284XSQ:1#%
```

ACLI's advanced grep is a more "intelligent" grep which tries to infer which lines of output are banner lines (which we want to preserve) and which lines are actual data records (which we want to filter based on our grep string). Advanced grep is invoked simply by using a double pipe '||':

```
VSP-8284XSQ:1#% show interfaces gigabitEthernet name || up
================================================================================
                                    Port Name
================================================================================
PORT                                          OPERATE  OPERATE  OPERATE
NUM      NAME                    DESCRIPTION   STATUS   DUPLX    SPEED    VLAN
--------------------------------------------------------------------------------
1/1                              10GbNone      up       full     10000    Acces
1/2                              10GbNone      up       full     10000    Acces
1/3                              10GbNone      up       full     10000    Acces
VSP-8284XSQ:1#%
```

Not only does advanced grep preserve the banner lines, but also if the output ends with a summary line informing the user about how many data records were displayed by the show command, these will also be preserved. In addition to that, ACLI's advanced grep is able to complement these summary lines with an extra line, inserted by ACLI, with the actual number of lines with data which were displayed after applying grep filtering. These summary lines are typically displayed on show commands which dump the MAC tables, ARP cache, IP routes, etc..:

```
VSP8400-3:1#% fdb ||00:dd:a0
        alias% show vlan mac-address-entry ||00:dd:a0
================================================================================
                                    Vlan Fdb
================================================================================
VLAN      MAC                                 SMLT
ID    STATUS    ADDRESS             INTERFACE   REMOTE   TUNNEL
--------------------------------------------------------------------------------
100   learned   00:dd:a0:02:01:00   Port-3/18   false    VSP8400-2
100   learned   00:dd:a0:02:02:00   c100:2/5    false    -
```

```
    c: customer vid    u: untagged-traffic
    126 out of 126 entries in all fdb(s) displayed.
    acli.pl: Displayed Record Count = 2
    Total peer MAC move count : 0
    VSP8400-3:1#%
```

Notice the lines above starting with *acli.pl:*; this was added by ACLI to indicate the number of records displayed by ACLI.

The other most flagrant case where simple line grep falls short is when trying to apply grep filtering to the switch config file, on Extreme devices which use a Cisco-like configuration syntax, where the config settings can be broken up into settings within a configuration context. For example, to enable an Ethernet port on VOSS or ERS platforms, it is necessary to first enter the Ethernet port configuration context with *'interface gigabitEthernet 1/1'* and then use a second command/line to enable the port *'no shutdown'*. A simple line grep for the port number (1/1) will yield the configuration context line, but not the *'no shutdown'* line; while a grep for *'shutdown'* will yield the *'no shutdown'* line, but not the interface configuration context line:

```
    VSP-8284XSQ:1#% cfg | 1/1
            alias% show running-config -ib |1/1
    interface GigabitEthernet 1/1

    VSP-8284XSQ:1#%
```

The above is not hugely useful, as we see none of the configuration pertaining to port 1/1.

```
    VSP-8284XSQ:1#% cfg | shutdown
            alias% show running-config -ib | shutdown
      no shutdown
      no shutdown
      no shutdown
      no shutdown

    VSP-8284XSQ:1#%
```

The above is just plain stupid. Ok, clearly 4 Ethernet ports on this switch are enabled, but which ones ?

If instead we use ACLI's advanced grep we obtain the following:

```
    VSP-8284XSQ:1#% cfg || 1/1
            alias% show running-config -ib || 1/1
    config terminal
    interface GigabitEthernet 1/1
       no shutdown
    exit
    qos queue-profile 1 member add 1/1-1/42,2/1-2/42
    end

    VSP-8284XSQ:1#%
```

Now we have something useful: all the configuration which pertains to port 1/1

```
    VSP-8284XSQ:1#% cfg || shutdown
            alias% show running-config -ib || shutdown
    config terminal
    interface GigabitEthernet 1/1
       no shutdown
    exit
    interface GigabitEthernet 1/2
       no shutdown
    exit
    interface GigabitEthernet 1/3
       no shutdown
    exit
    interface GigabitEthernet 1/4
       no shutdown
    exit
    end
```

```
        VSP-8284XSQ:1#%
```

And in the above case, we now get to see all enabled ports (configured with 'no shutdown') on the switch.

Note that ACLI's advanced grep becomes most compelling when the config file is properly indented, as in the examples above. However indentation of config files is not a given on all Extreme Network devices. On PassportERS/VOSS and BaystackERS platforms no indentation is provided and here it is necessary to include the '-i' switch after the "show running-config" command so that ACLI can apply the correct indentation. On other Extreme platforms (like the WLAN9100, SecureRouter, ISW), which also have a context-based configuration file, the indentation is already provided by the switch, so in this case the '-i' switch is not required (no harm adding it though). In general if using the preconfigured 'cfg' alias this will automatically produce the correct CLI command to display the switch configuration file with the '-i' & '-b' switches added accordingly.

In the case of other Extreme Networks devices which use a flat configuration file without any configuration contexts, ACLIs advanced grep does not bring any immediate benefits, as on these devices every line in the configuration file is fully contained and independent from any other line in the configuration file. Such is the case on ExtremeXOS switches and also on older PassportERS chassis using the old PPCLI. If we take an XOS switch, filtering on disabled ports works equally well with simple and advanced grep:

```
        EXOS-VM.11 #% cfg | disable port
                alias% show configuration -b | disable port
disable port 2
disable port 3
EXOS-VM.11 #%
EXOS-VM.11 #% cfg || disable port
                alias% show configuration -b || disable port
disable port 2
disable port 3
EXOS-VM.11 #%
```

And if we filter on a given port number:

```
        EXOS-VM.11 #% cfg | 2
                alias% show configuration -b | 2
disable port 2
configure ports 2 debounce time 0
EXOS-VM.11 #%
EXOS-VM.11 #% cfg || 2
                alias% show configuration -b || 2
configure vr VR-Default delete ports 1-3
configure vr VR-Default add ports 1-3
disable port 2
configure vlan Default add ports 1-3 untagged
configure ports 2 debounce time 0
EXOS-VM.11 #%
```

Yet, we see a difference in the above example when using advanced grep, which constitutes another benefit of using ACLI's advanced grep. Ethernet switches can have many (even hundreds of) ports. It is common to use short hand port ranges in the configuration files. However, if performing a simple grep on a port number which falls inside one of these ranges then the corresponding configuration line will be missed. The ACLI terminal, in interactive mode, knows exactly the port layout of the switch it is connected to (discovery of the port layout is done during initial connection/discovery). ACLI's advanced grep is able to leverage this information every time a port range is found in the configuration file and is thus able to determine whether or not the config line containing the port range should be displayed based upon the grep string provided. Follows a similar example with VOSS:

```
        VSP-8284XSQ:1#% cfg||1/2
                alias% show running-config -ib ||1/2
config terminal
vlan members remove 1 1/1-1/4
vlan members 10 1/1-1/4 portmember
interface GigabitEthernet 1/2
   no shutdown
exit
```

```
        qos queue-profile 1 member add 1/1-1/42,2/1-2/42
        end

        VSP-8284XSQ:1#%
```

Note that port 1/2 is included in the range of port members removed from default VLAN 1 and in the port range added to VLAN10; also it is part of QoS queue-profile range.

Regular expressions can of course also be used with ACLI's advanced grep.

Concentrating on using advanced grep on the connected device's configuration file, there are other configuration components where it is desirable to have a more intelligent grep function. The best example for an Ethernet switch is the VLAN construct which is always a major component of the configuration file. Would it not be desirable to perform a grep for one particular VLAN and obtain all the configuration lines for that VLAN ?. The ACLI advanced grep does just that when the grep string provided is in the format *vlan <id|list|range>*

```
        VSP8200-1:1#% cfg || vlan 210
              alias% show running-config -ib || vlan 210
        config terminal
        vlan create 210 name "Green-Cmp210" type port-mstprstp 0
        vlan i-sid 210 2800210
        interface Vlan 210
           vrf green
           ip address 20.1.210.1 255.255.255.0 2
           ip spb-multicast enable
           ip dhcp-relay
           ip rsmlt
           ip rsmlt holdup-timer 9999
        exit
        router vrf green
           ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200
           ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200  enable
           ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200  mode bootp_dhcp
        exit
        ip rsmlt peer-address 20.1.210.2 b0:ad:aa:42:95:02 210
        slpp vid  210
        end

        VSP8200-1:1#%
```

Note that not all the lines returned have an exact match for the string *'vlan 210'*. ACLI's advanced grep is being a bit more clever in knowing what lines are likely to be applicable to the configuration of VLAN 210 on, in this case, a VOSS switch. Not only, some lines, like the dhcp-relay ones, have absolutely no reference to any VLAN, so how does ACLI know they need to be included? The trick is that these lines contain the IP address which is configured on VLAN 210 and therefore are included in the final result.

Similar to the VLAN example, ACLI's advanced grep offers grep for a range of other configuration components. The embedded *'@help'* command gives a complete list:

```
        show running ||vlan [<vid-or-name-list>]          grep on vlan config contexts
        show running ||mlt [<mltid-list>]                 grep on mlt config contexts
        show running ||loopback [<CLIPid-list>]           grep on loopback config contexts
        show running ||<ospf|rip|isis|bgp list>           grep on well known router config cont
        show running ||router <protocol-list>             grep on any router config context
        show running ||vrf [<vrfname-list>]               grep on router vrf config contexts
        show running ||route-map [<routemap-list>]        grep on route-map config contexts
        show running ||i-sid [<isid-list>]                grep on i-sid config contexts
        show running ||acl [<acl-id-list>]                grep on filter acl ids
        show running ||logical-intf [<intf-list>]         grep on ISIS logical interfaces
        show running ||mgmt [<port-list or id-list>]      grep on Mgmt interfaces
        show running ||ssid [<ssid-name-list>]            grep on WLAN SSIDs
```

Some examples follow.

Grep on MLTs:

```
VSP8200-1:1#% cfg || mlt 1,14
        alias% show running-config -ib || mlt 1,14
config terminal
mlt 1 enable name "ERS4900-STK"
mlt 1 member 2/1
mlt 1 encapsulation dot1q
mlt 14 enable name "X670-4"
mlt 14 member 2/14
mlt 14 encapsulation dot1q
interface mlt 1
    smlt
    fa
    fa enable
    fa management i-sid 2800209 c-vid 209
exit
interface mlt 14
    smlt
    fa
    fa enable
    no fa message-authentication
    fa management i-sid 2800209 c-vid 209
exit
end

VSP8200-1:1#%
```

Grep on router instance:

```
VSP8200-1:1#% cfg || router isis
        alias% show running-config -ib || router isis
config terminal
router isis
    spbm 1
    spbm 1 nick-name 0.00.31
    spbm 1 b-vid 4051-4052 primary 4051
    spbm 1 multicast enable
    spbm 1 ip enable
    spbm 1 ipv6 enable
    spbm 1 smlt-virtual-bmac 82:bb:00:00:31:ff
    spbm 1 smlt-peer-system-id 82bb.0000.3200
exit
router isis
    sys-name "VSP8200-1"
    ip-source-address 20.0.20.31
    ip-tunnel-source-address 172.16.0.81 vrf fe
    ipv6-source-address 2000:20:0:0:0:0:0:31
    is-type l1
    system-id 82bb.0000.3100
    manual-area 49.0000
exit
router isis enable
router isis
    accept i-sid 3800009 enable
exit
router isis
    redistribute direct
    redistribute direct route-map "Suppress-IST"
    redistribute direct enable
exit
end

VSP8200-1:1#%
```

Grep on VRF instance:

```
VSP8200-1:1#% cfg || vrf green
        alias% show running-config -ib || vrf green
config terminal
ip vrf green vrfid 1
interface Vlan 210
    vrf green
```

```
          exit
          router vrf green
             ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200
             ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200   enable
             ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200   mode bootp_dhcp
          exit
          router vrf green
             ipvpn
             i-sid 3800001
             mvpn enable
             ipvpn enable
          exit
          router vrf green
             isis accept i-sid 3800009 enable
          exit
          router vrf green
             isis redistribute direct
             isis redistribute direct enable
          exit
          isis apply redistribute direct vrf green
          end

          VSP8200-1:1#%
```

Grep of route-map:

```
          VSP8400-3:1#% cfg || route-map host-routes
                alias% show running-config -ib || route-map host-routes
          config terminal
          router vrf shared
             route-map "host-routes" 1
                no permit
                enable
                match network "host-routes"
             exit
          exit
          router vrf shared
             isis accept route-map "host-routes"
             isis accept isid-list "client-vrfs" route-map "host-routes"
          exit
          end

          VSP8400-3:1#%
```

Advanced Negative Grep

Advanced negative grep is very much the same as advanced grep, except that it will hide the lines matching the negative advanced grep instead of showing them. Negative advanced grep is invoked simply by using a double bang '!!'. So, if we have a switch with this simple config and just one VLAN10:

```
VSP-8284XSQ:1#% cfg
         alias% show running-config -ib
config terminal
boot config flags sshd
boot config flags telnetd
password password-history 3
ssh
no web-server secure-only
vlan members remove 1 1/1-1/4
vlan create 10 type port-mstprstp 0
vlan members 10 1/1-1/4 portmember
interface Vlan 10
   ip address 192.168.10.1 255.255.255.0 0
exit
interface mgmtEthernet mgmt
   auto-negotiate
   ip address 192.168.56.71 255.255.255.0
exit
qos queue-profile 1 member add 1/1-1/42,2/1-2/42
no ntp
slpp enable
slpp vid  10
end

VSP-8284XSQ:1#%
```

A positive advanced grep for VLAN10 yields:

```
VSP-8284XSQ:1#% cfg||vlan 10
         alias% show running-config -ib ||vlan 10
config terminal
vlan create 10 type port-mstprstp 0
vlan members 10 1/1-1/4 portmember
interface Vlan 10
   ip address 192.168.10.1 255.255.255.0 0
exit
slpp vid  10
end

VSP-8284XSQ:1#%
```

While a negative advanced grep for VLAN10 yields:

```
VSP-8284XSQ:1#% cfg!!vlan 10
         alias% show running-config -ib !!vlan 10
config terminal
boot config flags sshd
boot config flags telnetd
password password-history 3
ssh
no web-server secure-only
vlan members remove 1 1/1-1/4
interface mgmtEthernet mgmt
   auto-negotiate
   ip address 192.168.56.71 255.255.255.0
exit
qos queue-profile 1 member add 1/1-1/42,2/1-2/42
no ntp
slpp enable
end
```

```
VSP-8284XSQ:1#%
```

Negative advanced grep often makes more sense when concatenated with positive greps so as to remove some unwanted output from the output produced from the previous grep function.

Let us take the same advanced positive grep of VLAN 210 as we did before:

```
VSP8200-1:1#% cfg || vlan 210
        alias% show running-config -ib || vlan 210
config terminal
vlan create 210 name "Green-Cmp210" type port-mstprstp 0
vlan i-sid 210 2800210
interface Vlan 210
   vrf green
   ip address 20.1.210.1 255.255.255.0 2
   ip spb-multicast enable
   ip dhcp-relay
   ip rsmlt
   ip rsmlt holdup-timer 9999
exit
router vrf green
   ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200
   ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200  enable
   ip dhcp-relay fwd-path 20.1.210.1 20.9.190.200  mode bootp_dhcp
exit
ip rsmlt peer-address 20.1.210.2 b0:ad:aa:42:95:02 210
slpp vid  210
end

VSP8200-1:1#%
```

And let's assume we do not want to see the DHCP and RSMLT and SLPP config sections of the VLAN 210; so we can simply append a negative grep to suppress the those lines:

```
VSP8200-1:1#% cfg || vlan 210 !! dhcp,rsmlt,slpp
        alias% show running-config -ib || vlan 210 !! dhcp,rsmlt,slpp
config terminal
vlan create 210 name "Green-Cmp210" type port-mstprstp 0
vlan i-sid 210 2800210
interface Vlan 210
   vrf green
   ip address 20.1.210.1 255.255.255.0 2
   ip spb-multicast enable
exit
end

VSP8200-1:1#%
```

Offline Grep

The advanced grep capability of the ACLI terminal can get rather addictive, in particular when looking and dicing switch configuration files! In order to use the same capability on offline configuration files, the ACLI Perl executable can be used in a DOS/Cmd/Shell window where offline configuration files can be fed to it with the desired grep string. The ACLI *-g* command line switch is required. There are two ways to use this capability:

- Pipe the configuration file from some other command/executable into ACLI

        % cat myconfig.cfg | acli -g "-ib ||spbm"

- Let ACLI open the file(s) directly

        % acli -g "-ib ||spbm" myconfig1.cfg [myconfig2.cfg] ...

    This form also allows wildcards to be used for the input config files; for example, to pick up all *.cfg files in the same directory:

        % acli -g "-ib ||spbm" *.cfg

The *-g* argument takes this format:

        -g "[-ib <grep-mode: |,!,||,!!|^>] <grep-string> [<grep-mode2> <grep-string2>] ..."

The *-i* & *-b* switches are the usual ACLI switches which enable indentation and remove comment lines from the configuration file.

The *grep-mode* is one of the usual ACLI grep mode: simple grep '|', simple negative grep '!', advanced grep '||', advanced negative grep '!!', or highlight '^'

Multiple greps can be concatenated as usual.

If the very first *grep-mode* is omitted, as well as the -ib switches, then ACLI automatically pre-pends '-ib ||' to the first *grep-string* provided, since this is the most common use case.

ACLI's grep requires knowledge of the correct switch family type in order to function correctly. When online, the family type of the connected switch can be inferred, but offline this may not be possible. The current grep offline behaviour is that BaystackERS family type is automatically detected from ERS stackable config files and otherwise the family type is assumed to be PassportERS/VOSS. To use offline grep with other switch family types, you will need to set this using the ACLI *-f* command line switch: *-f <type>*

A few examples to see ACLI's offline grep in action:

        C:\>acli -g "-ib ||spbm" SA01-config.cfg
        config terminal
        spbm
        spbm ethertype 0x8100
        router isis
           spbm 1
           spbm 1 nick-name 5.04.01
           spbm 1 b-vid 4048-4049 primary 4048
           spbm 1 multicast enable
           spbm 1 ip enable

```
        exit
        vlan create 4048 name "BB-VLAN_4048" type spbm-bvlan
        vlan create 4049 name "BB-VLAN_4049" type spbm-bvlan
        interface GigabitEthernet 1/49
           isis spbm 1
           isis spbm 1 l1-metric 2000
        exit
        interface GigabitEthernet 1/50
           isis spbm 1
           isis spbm 1 l1-metric 2000
        exit
        cfm spbm mepid 5042
        cfm spbm enable
        end
```

Or we can get exactly the same output with just:

```
        C:\>acli -g "spbm" SA01-config.cfg
        config terminal
        spbm
        spbm ethertype 0x8100
        router isis
           spbm 1
           spbm 1 nick-name 5.04.01
           spbm 1 b-vid 4048-4049 primary 4048
           spbm 1 multicast enable
           spbm 1 ip enable
        exit
        vlan create 4048 name "BB-VLAN_4048" type spbm-bvlan
        vlan create 4049 name "BB-VLAN_4049" type spbm-bvlan
        interface GigabitEthernet 1/49
           isis spbm 1
           isis spbm 1 l1-metric 2000
        exit
        interface GigabitEthernet 1/50
           isis spbm 1
           isis spbm 1 l1-metric 2000
        exit
        cfm spbm mepid 5042
        cfm spbm enable
        end
```

And the same, by piping the config file to ACLI via STDIN (on a Unix system, you would use 'cat' not 'type')

```
        C:\>type SA01-config.cfg | acli -g spbm
        config terminal
        spbm
        spbm ethertype 0x8100
        router isis
           spbm 1
           spbm 1 nick-name 5.04.01
           spbm 1 b-vid 4048-4049 primary 4048
           spbm 1 multicast enable
           spbm 1 ip enable
        exit
        vlan create 4048 name "BB-VLAN_4048" type spbm-bvlan
        vlan create 4049 name "BB-VLAN_4049" type spbm-bvlan
        interface GigabitEthernet 1/49
           isis spbm 1
           isis spbm 1 l1-metric 2000
        exit
        interface GigabitEthernet 1/50
           isis spbm 1
           isis spbm 1 l1-metric 2000
        exit
        cfm spbm mepid 5042
```

```
       cfm spbm enable
       end
```

Let's get the SPB nick-name config across a bunch of offline config files:

```
C:\>acli -g "nick-name" *.cfg


SA01-config.cfg:
================
config terminal
router isis
   spbm 1 nick-name 5.04.01
exit
end


SA02-config.cfg:
================
config terminal
router isis
   spbm 1 nick-name 5.04.02
exit
end


SA03-config.cfg:
================
config terminal
router isis
   spbm 1 nick-name 5.04.03
exit
end


SA04-config.cfg:
================
config terminal
router isis
   spbm 1 nick-name 5.04.04
exit
end


SA05-config.cfg:
================
config terminal
router isis
   spbm 1 nick-name 5.04.05
exit
end


SA06-config.cfg:
================
config terminal
router isis
   spbm 1 nick-name 5.04.06
exit
end


SA07-config.cfg:
================
config terminal
router isis
   spbm 1 nick-name 5.04.07
```

```
        exit
        end


        SA08-config.cfg:
        ================
        config terminal
        router isis
            spbm 1 nick-name 5.04.08
        exit
        end


        SD01-config.cfg:
        ================
        config terminal
        router isis
            spbm 1 nick-name 5.04.d1
        exit
        end


        SD02-config.cfg:
        ================
        config terminal
        router isis
            spbm 1 nick-name 5.04.d2
        exit
        end
```

Perhaps, in the above case, a simple grep would give more compact output:

```
        C:\>acli -g "|nick-name" *.cfg


        SA01-config.cfg:
        ================
        spbm 1 nick-name 5.04.01


        SA02-config.cfg:
        ================
        spbm 1 nick-name 5.04.02


        SA03-config.cfg:
        ================
        spbm 1 nick-name 5.04.03


        SA04-config.cfg:
        ================
        spbm 1 nick-name 5.04.04


        SA05-config.cfg:
        ================
        spbm 1 nick-name 5.04.05


        SA06-config.cfg:
        ================
        spbm 1 nick-name 5.04.06


        SA07-config.cfg:
        ================
```

```
        spbm 1 nick-name 5.04.07


        SA08-config.cfg:
        ================
        spbm 1 nick-name 5.04.08


        SD01-config.cfg:
        ================
        spbm 1 nick-name 5.04.d1


        SD02-config.cfg:
        ================
        spbm 1 nick-name 5.04.d2
```
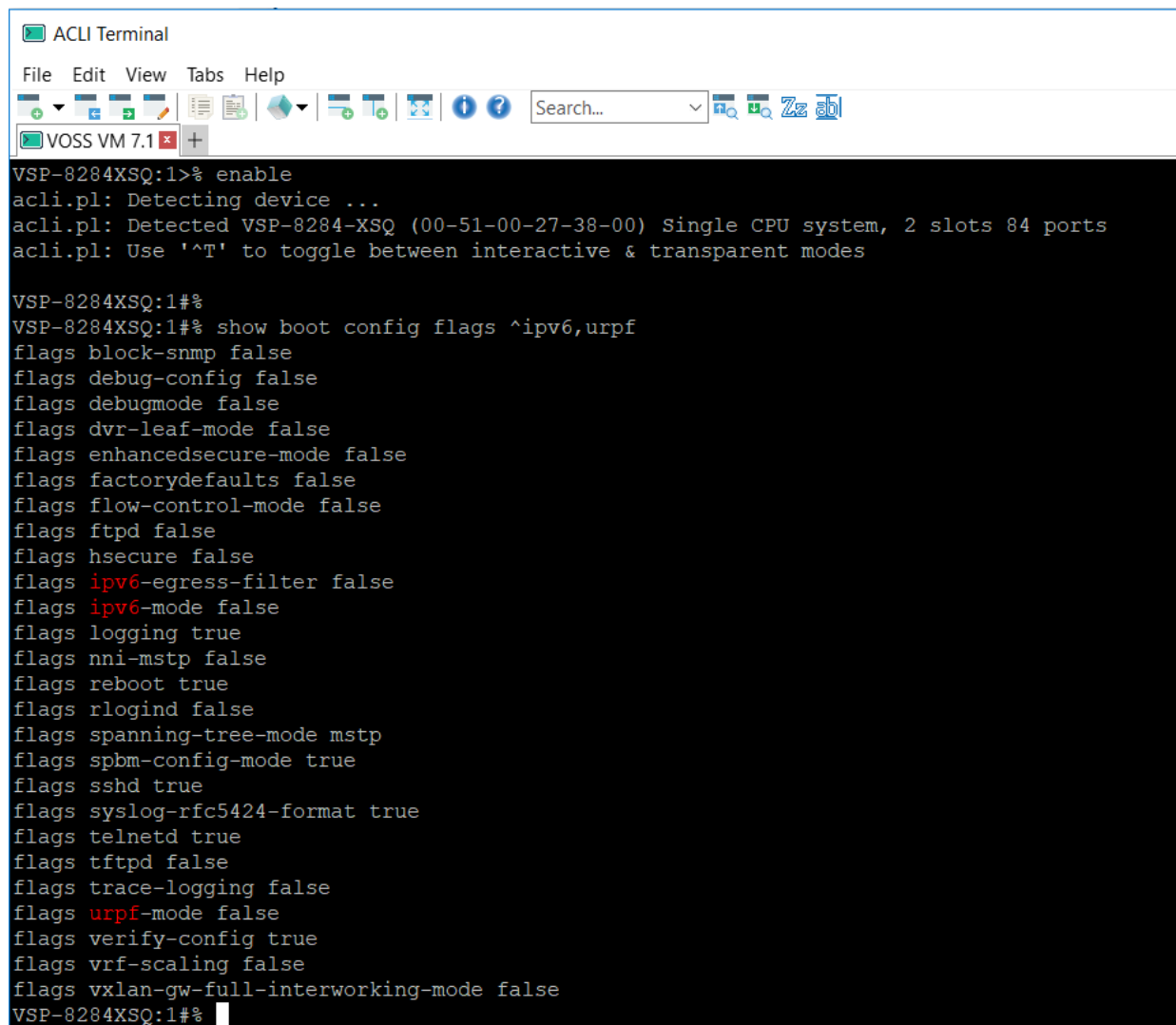
If a config file produces no output, then it is not listed.

Another way of working with offline configuration files is to use ACLI's pseudo mode; refer to the Pseudo mode section.

Highlighting Output

It is often hard to spot relevant information when a CLI command produces a lot of output. Of course grep can be used to filter out only lines matching a desired pattern, but in some cases lines preceding and/or following a given pattern can be equally valuable. A good example is when dumping the device's log file for a specific event and any other events leading up to it, or deriving from it.

The ACLI terminal, in interactive mode, is capable of highlighting text in the output. To do so simply append to any command the '^' character followed by the patterns to match and highlight: ^ *<pattern>*

```
ACLI Terminal
File  Edit  View  Tabs  Help

VOSS VM 7.1 ✕  +

VSP-8284XSQ:1>% enable
acli.pl: Detecting device ...
acli.pl: Detected VSP-8284-XSQ (00-51-00-27-38-00) Single CPU system, 2 slots 84 ports
acli.pl: Use '^T' to toggle between interactive & transparent modes

VSP-8284XSQ:1#%
VSP-8284XSQ:1#% show boot config flags ^ipv6,urpf
flags block-snmp false
flags debug-config false
flags debugmode false
flags dvr-leaf-mode false
flags enhancedsecure-mode false
flags factorydefaults false
flags flow-control-mode false
flags ftpd false
flags hsecure false
flags ipv6-egress-filter false
flags ipv6-mode false
flags logging true
flags nni-mstp false
flags reboot true
flags rlogind false
flags spanning-tree-mode mstp
flags spbm-config-mode true
flags sshd true
flags syslog-rfc5424-format true
flags telnetd true
flags tftpd false
flags trace-logging false
flags urpf-mode false
flags verify-config true
flags vrf-scaling false
flags vxlan-gw-full-interworking-mode false
VSP-8284XSQ:1#%
```

Regular expressions can of course be used as the match pattern.

By default the highlight is bright red. It can be customized to other standard
ANSI colours using the embedded *@highlight* command or the *'highlight'*
command under the ACLI control interface. Brightness, background, reverse
and underline can also be customized.

```
VSP-8284XSQ:1#% @highlight info

        Highlight foreground : red
        Highlight background : disabled
        Highlight bright     : enabled
        Highlight reverse    : disabled
        Highlight underline  : disabled
        Highlight rendering  : SAMPLE

VSP-8284XSQ:1#%
```

All these settings can also be modified in the acli.ini file using keys:

- *highlight_fg_colour_str*
- *highlight_bg_colour_str*
- *highlight_text_bright_flg*
- *highlight_text_underline_flg*
- *highlight_text_reverse_flg*

Note that if, under ConsoleZ Edit / Settings / Appearance / Font, you enable a "Custom color", the ACLI highlight capability will appear to not work anymore. This is because ConsoleZ is now re-colouring all output in the window to the new custom colour you have set there. If you want to change your default font colour, you should do this under the setting of your system's cmd window settings instead. To do this run a "cmd" regular DOS box, then right-click the window banner and select properties; select the colours tab and set the "screen text" colour here (ConsoleZ will then use that as default font colour and the highlighting feature will still work).

C:\WINDOWS\system32\cmd.exe

Microsoft Windows [Version 10.0.16299.431]
(c) 2017 Microsoft Corpor...

C:\Users\lstevens>

"C:\WINDOWS\system32\cmd.exe" Properties

| Options | Font | Layout | Colors |

○ Screen Text
○ Screen Background
○ Popup Text
○ Popup Background

Selected Color Values

Red: 0
Green: 0
Blue: 0

Selected Screen Colors

```
C:\WINDOWS> dir
SYSTEM        <DIR>        10-01-99    5:00a
SYSTEM32      <DIR>        10-01-99    5:00a
```

Selected Popup Colors

```
C:\WINDOWS> dir
SYSTEM        <DIR>        10-01-99    5:00a
SYSTEM32      <DIR>        10-01-99    5:00a
```

Opacity

30%   100%   100

OK   Cancel

Stream Editor (SED)

Stream editing (named sed as per the Unix utility) allows output from the switch or input to the switch to be modified and edited on the fly. The most common use of this feature will be either to modify the output of the switch or to re-colour certain keywords or addressing identifiers. Manipulation of the input stream is also possible but not recommended.

For both input and output, the feature allows a pattern match string to be supplied with a corresponding replacement string; this is implemented using Perl's *s/PATTERN/REPLACEMENT/mgee* operator, so capturing parentheses are allowed in the regex PATTERN and can be re-used in the REPLACEMENT string. The REPLACEMENT string can also be a Perl code snippet. For output only, it is also possible to associate a pattern with a re-colouring profile. Re-colouring is implemented using *s/PATTERN/<start-ANSI-colour-sequence>$&<stop-ANSI-colour-sequence>/mgee*; when defining re-colouring patterns it will come useful to polish up on Perl's lookbehind and lookahead assertions as well as code assertions which can allow matching patterns immediately before or immediately after the actual pattern we want to recolour without recolouring those as well (refer to [Perldoc perlre](#)). Only the 8 standard colours are supported, for either foreground or background as well as the ability to set any of bright, underline and reverse.

The sed patterns can be set either manually via the embedded *'@sed'* command or via the *'sed'* command in the ACLI control interface or or they can be setup in the *acli.sed* file.

We shall look at the former first. A new pattern can be set with one of the following syntaxes:

```
@sed input add [<index:1-20>] '<pattern>' '<replacement>'
@sed input add [<index:1-20>] '<pattern>' '{<replacement-code>}'
@sed output add [<index:1-20>] '<pattern>' '<replacement>'
@sed output add [<index:1-20>] '<pattern>' '{<replacement-code>}'
@sed output add colour [<index:1-20>] '<pattern>' '<colour-profile>'
```

In all cases an index for the pattern must be supplied (except in scripting mode) which will determine the order in which the sed patterns will be applied. ACLI currently limits the patterns to a maximum of 20 per pattern type (this can be overridden in *'acli.sed'*) as too many patterns might start consuming too much CPU. In scripting/sourcing mode it is possible to omit the index on the above commands and additional patterns will be accepted with an auto-generated index starting from 21 (or the overridden value + 1 specified in *'acli.sed'*). Sed patterns defined in sourcing mode (with index > 20) will not be persistent and will automatically disappear when existing sourcing mode.

The *'<pattern>' & '<replacement>'* fields need to be quoted and will be used in the Perl operator *s/<pattern>/<replacement>/mgee*

The *'{<replacement-code>}'* field need to be quoted in single quotes + enclosed in curlies {} and will be used in the Perl operator *s/<pattern>/&{<replacement-code>}/mge*

The last syntax defines a pattern to be output re-coloured and the *<colour-profile>* needs to refer to an already existing sed colour profile.

All patterns are either assigned globally (will be used againsat any Extreme device in interactive mode) or for a specific Extreme product family type. The above *'@sed'* commands will always apply the pattern to the currently active product type. Defining global patterns needs to be done in the *acli.sed* file or by using the *sed* command in the ACLI control interface when the terminal is in transparent mode.

Sed colour profiles can be created with the following syntax:

```
@sed colour <profile-name> background|bright|foreground|reverse|underline
```

Where *background|foreground* can be set to any of: *black|blue|cyan|green|magenta|none|red|white|yellow*; and *bright|reverse|underline* can be either *enabled* or *disabled*

Let us see a simple example of how we can use this feature. Here is the log file of a typical VOSS switch:

```
VSP-8284XSQ:1#% show log file
        1 2020-01-19T13:12:09.198Z VSP-8284XSQ IO1 - 0x00270428 - 00000000 GlobalRouter SW INFO I
        1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:09.252Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:10.770Z VSP-8284XSQ IO1 - 0x00264541 - 00000000 GlobalRouter SW INFO I
        1 2020-01-19T13:12:10.780Z VSP-8284XSQ IO1 - 0x0026452f - 00000000 GlobalRouter SW INFO N
        1 2020-01-19T13:12:11.253Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.256Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.258Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.259Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.259Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.264Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.276Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.289Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.295Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.298Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.309Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.322Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.323Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
        1 2020-01-19T13:12:11.332Z VSP-8284XSQ IO1 - 0x0027042b - 00000000 GlobalRouter SW INFO P
```

These lines can often be very long and some of the information provided in these lines could be omitted; for instance the *00000000* number is not hugely useful; let's remove it with the following sed pattern:

```
VSP-8284XSQ:1#% @sed output add 1 '- 00000000 ' ''
Output pattern 1 added : '- 00000000 ' => ''
```

If we now show the log file we get:

```
VSP-8284XSQ:1#% show log file
1 2020-01-19T13:12:09.198Z VSP-8284XSQ IO1 - 0x00270428 GlobalRouter SW INFO Lifecycle: S
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process namS
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process sock
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process oom9
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process oom9
1 2020-01-19T13:12:09.252Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process imgs
1 2020-01-19T13:12:10.770Z VSP-8284XSQ IO1 - 0x00264541 GlobalRouter SW INFO Image Integr
1 2020-01-19T13:12:10.780Z VSP-8284XSQ IO1 - 0x0026452f GlobalRouter SW INFO No patch set
1 2020-01-19T13:12:11.253Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process logS
1 2020-01-19T13:12:11.256Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process trcS
1 2020-01-19T13:12:11.258Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process oobS
1 2020-01-19T13:12:11.259Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process nick
1 2020-01-19T13:12:11.259Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process nick
1 2020-01-19T13:12:11.264Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process hwsS
1 2020-01-19T13:12:11.276Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process redi
1 2020-01-19T13:12:11.289Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process cbcp
1 2020-01-19T13:12:11.295Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process rssS
1 2020-01-19T13:12:11.298Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process dbgS
1 2020-01-19T13:12:11.309Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process dbgS
1 2020-01-19T13:12:11.322Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process khiC
1 2020-01-19T13:12:11.323Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process core
1 2020-01-19T13:12:11.332Z VSP-8284XSQ IO1 - 0x0027042b GlobalRouter SW INFO Process file
```

This is better. But we could go further. The reference to *GlobalRouter* is also of little value. Let's remove that as well:

```
VSP-8284XSQ:1#% @sed output add 2 'GlobalRouter ' ''
Output pattern 2 added : 'GlobalRouter ' => ''
```

And we get a much more compact log file:

```
VSP-8284XSQ:1#% show log file
1 2020-01-19T13:12:09.198Z VSP-8284XSQ IO1 - 0x00270428 SW INFO Lifecycle: Start
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process namServer started
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process sockserv started,
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process oom95 started, pi
1 2020-01-19T13:12:09.251Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process oom90 started, pi
1 2020-01-19T13:12:09.252Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process imgsync.x started
1 2020-01-19T13:12:10.770Z VSP-8284XSQ IO1 - 0x00264541 SW INFO Image Integrity verificat
1 2020-01-19T13:12:10.780Z VSP-8284XSQ IO1 - 0x0026452f SW INFO No patch set.
1 2020-01-19T13:12:11.253Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process logServer started
1 2020-01-19T13:12:11.256Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process trcServer started
1 2020-01-19T13:12:11.258Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process oobServer started
1 2020-01-19T13:12:11.259Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process nickServer starte
1 2020-01-19T13:12:11.259Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process nickClient starte
1 2020-01-19T13:12:11.264Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process hwsServer started
1 2020-01-19T13:12:11.276Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process redis-server star
1 2020-01-19T13:12:11.289Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process cbcp-main.x start
1 2020-01-19T13:12:11.295Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process rssServer started
1 2020-01-19T13:12:11.298Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process dbgServer started
1 2020-01-19T13:12:11.309Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process dbgShell started,
1 2020-01-19T13:12:11.322Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process khiCollection sta
1 2020-01-19T13:12:11.323Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process coreManager.x sta
1 2020-01-19T13:12:11.332Z VSP-8284XSQ IO1 - 0x0027042b SW INFO Process filer started, pi
```

These sed patterns could be permanently defined in the *'acli.sed'* file and get applied to all output received from the switch. Though in the above example it would probably make more sense to add the relevant *@sed* commands to the alias command used to display the log file, then these will only apply to the output of the "show log file" command and nothing else. The *'log'* alias in the default supplied *acli.alias* file now does precisely that and, for VOSS devices, executes the following commands:

```
@sed output add '- 00000000 |GlobalRouter ' ''; show log file
```

Note that since the replacement patterns are the same (empty string) we can combine the patterns into one, which is more efficient. Also a semi-colon fragmented command is in effect an ACLI script so the individual commands will be executed in sourcing mode, which means we do not have to set an index number for the pattern so that the pattern will be dynamic and will automatically get deleted once ACLI comes out of sourcing mode, i.e. once the alias command execution has completed.

Now let's see an example where we use sed patterns to recolour the output. These sed patterns define 2 colours, green and red, and assign these to any occurrence of "up" or "down" respectively:

```
VSP8200-1:1#% @sed colour green foreground green
VSP8200-1:1#% @sed colour green bright enable
VSP8200-1:1#% @sed output add colour 3 '(?i)\bup\b' green
Output pattern 1 added : '(?i)\bup\b' => colour green

VSP8200-1:1#% @sed colour red foreground red
VSP8200-1:1#% @sed colour red bright enable
VSP8200-1:1#% @sed output add colour 4 '(?i)\bdown\b' red
Output pattern 2 added : '(?i)\bdown\b' => colour red
```

We now get output colored as shown:

To view currently defined sed colour profiles and patterns use the following:

```
@sed colour info
@sed info
```


Note how the colouring sed replacement pattern simply pre-pends the text to re-colour with the corresponding ANSI escape sequence for the desired colour and then appends another ANSI escape sequence to reset the colours back to default.

As mentioned before, input patterns are a bit more delicate. The following is an example used by the ACLI author which makes use of input patterns for an application where many VSPs are scripted via a single ACLI script. Each VSP gets the slot number of its ports re-mapped from 1 to a new value X (assigned to $u here) which reflects the rack number of where the VSP switch is located (kind of distributed stacking via ACLI!). The VSPs in question are VSP4000s which have all ports on slot-1, and after appling the following @sed config it will appear that the VSP has all its ports on slot-12:

```
$u = 12
@sed output add 1 '\b1/(\d)' "$u/\$1"
@sed input add 1 "\b$u/(\d)" '1/$1'
```

We get this:

```
VSP-8284XSQ:1#% $u = 12

$u              = 12

VSP-8284XSQ:1#% @sed output add 1 '\b1/(\d)' "$u/\$1"
```

```
            vars% @sed output add 1 '\b1/(\d)' "12/$1"
Output pattern 1 added : '\b1/(\d)' => '12/$1'

VSP-8284XSQ:1#% @sed input add 1 "\b$u/(\d)" '1/$1'
            vars% @sed input add 1 "\b12/(\d)" '1/$1'
Input pattern 1 added : '\b12/(\d)' => '1/$1'

VSP-8284XSQ:1#%
VSP-8284XSQ:1#% if
        alias% show interfaces gigabitEthernet interface!!\blocked
================================================================================================
                                  Port Interface
================================================================================================
PORT                            LINK  PORT           PHYSICAL           STATUS
NUM      INDEX DESCRIPTION       TRAP  LOCK   MTU     ADDRESS            ADMIN   OPERATE
------------------------------------------------------------------------------------------------
12/1     192   10GbNone          true  false  1950    00:51:00:3f:a8:00  down    down
12/2     193   10GbNone          true  false  1950    00:51:00:3f:a8:01  down    down
12/3     194   10GbNone          true  false  1950    00:51:00:3f:a8:02  down    down
12/4     195   10GbNone          true  false  1950    00:51:00:3f:a8:03  down    down
12/5     196   10GbNone          true  false  1950    00:51:00:3f:a8:04  down    down
12/6     197   10GbNone          true  false  1950    00:51:00:3f:a8:05  down    down
12/7     198   10GbNone          true  false  1950    00:51:00:3f:a8:06  down    down
12/8     199   10GbNone          true  false  1950    00:51:00:3f:a8:07  down    down
12/9     200   10GbNone          true  false  1950    00:51:00:3f:a8:08  down    down
12/10    201   10GbNone          true  false  1950    00:51:00:3f:a8:09  down    down
12/11    202   10GbNone          true  false  1950    00:51:00:3f:a8:0a  down    down
12/12    203   10GbNone          true  false  1950    00:51:00:3f:a8:0b  down    down
12/13    204   10GbNone          true  false  1950    00:51:00:3f:a8:0c  down    down
12/14    205   10GbNone          true  false  1950    00:51:00:3f:a8:0d  down    down
12/15    206   10GbNone          true  false  1950    00:51:00:3f:a8:0e  down    down
--More (q=Quit, space/return=Continue, ^P=Toggle on/off)--
```

And thanks to the input sed pattern, the illusion is complete, even if configuring those ports:

```
VSP-8284XSQ:1#% configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
VSP-8284XSQ:1(config)#% ife 12/1
                alias% interface gigabitEthernet 12/1
VSP-8284XSQ:1(config-if)#% name test
VSP-8284XSQ:1(config-if)#% ifname
                  alias% show interfaces gigabitEthernet name ||\d+/\d+\s+\S.*\S\s+\S+\
================================================================================================
                                    Port Name
================================================================================================
PORT                                              OPERATE  OPERATE  OPERATE
NUM      NAME                     DESCRIPTION      STATUS   DUPLEX   SPEED    VL
------------------------------------------------------------------------------------------------
12/1     test                     10GbNone         down     full     0        A
VSP-8284XSQ:1(config-if)#%
```

To delete a sed colour profile or a pattern use the following syntaxes:

```
@sed colour <profile-name> delete
@sed input delete <index:1-20>
@sed output delete <index:1-20>
```

To reset and remove all sed patterns use the following command:

```
@sed reset
```

A better way to define sed patterns and colour profiles is to supply them in the *acli.sed* file. A *acli.sed* file is shipped by default with ACLI and contains a number of pre-defined colour profiles and sed patterns and examples; this file is contained in the ACLI install directory and is a versioned file, which means it is liable to get updated and replaced when the ACLI update script is run, if a newer version of it exists.

If you'd like to edit or modify this file you should create your own *acli.sed* file in one of the following paths:

- ENV path *%ACLI%* (if you defined it)
- ENV path *$HOME/.acli* (on Unix systems)
- ENV path *%USERPROFILE%\.acli* (on Windows)

The *acli.sed* file needs to be edited with a specific syntax.

Lines commencing with '#' are comment lines and are ignored

Before any sed patterns are defined, a *start-id* and *max-id* directives can be set:

- A max-id can be supplied to override the default maximum value of 20 for pattern indexes which is otherwise hardcoded in ACLI terminal. This maximum value applies to each of the three possible sed pattern types; i.e. up to 20 output patters, 20 output re-colouring patterns and 20 input patterns. It is not a good idea to have too many sed patterns, this can affect performance. Up to 20 seems ok.

  ```
  max-id = <number>
  ```

- Every pattern is assigned an index number which determines the order in which the patterns are applied; patterns read in from this file will be applied with sequential indexes starting from *start-id*. By setting a value > 1 it is possible to reserve index numbers from 1 to X for interactive use via the @sed embedded command, while patterns defined in *acli.sed* will get allocated index numbers from X+1 to *max-id*.

  ```
  start-id = <number>
  ```

Colour profles are defined with the following syntax:

```
colour <profile-name> [foreground <black|blue|cyan|green|magenta|red|white|yellow>] [back
```

All patterns defined in the file (Input, Output and Output Colour) can be categorized as either global or against any of the Conrrol::CLI family types. This allows to reduce the number of patterns checked against any family device type. The order of patterns can however remain important, so the applicable category can be set at any time and all subsequent pattern definitions will apply to that category, until a new category is set. If no category is set, global will apply as default for all patterns. A list of family types can also be specified ("global" must not be in the list) in which case subsequent patterns are enumerated sequentially for each family type listed.

```
category global
category [list of: BaystackERS|PassportERS|ExtremeXOS|ISW|Series200|Wing|SLX|SecureRouter
```

While patterns can be supplied with these syntaxes:

```
[output] '<pattern>' colour '<profile-name>' [# <optional comments>]
[output] '<pattern>' '<replacement>' [# <optional comments>]
[output] '<pattern>' {<replacecode>} [# <optional comments>]
intput '<pattern>' '<replacement>' [# <optional comments>]
intput '<pattern>' {<replacecode>} [# <optional comments>]
```

Where

- *output* keyword is optional and can be omitted
- *input* keyword must be specified
- *<pattern>* regular expression pattern; must be enclosed in single or double quotes
- *<replacement>* replacement string; must also be enclosed in single or double quotes
- *<replacecode>* replacement code; must be enclosed in curlies {}
- *<profile-name>* must be a previuosly defined colour profile name from this file; can be quoted
- *<optional comments>* comments can be placed on same line after '#' character

Also refer to the default supplied *acli.sed* file for further syntax and examples.

To reload the *acli.sed* file on already running ACLI sessions the following embedded command can be used:

```
@sed reload
```

This will first delete all sed patterns, like *@sed reset* does, and then will reload the *acli.sed* file

Multiple Commands per line

On some historical Extreme Networks products (PassportERS native Passport/Accelar CLI) it was possible to enter and execute multiple commands on the same line, separated by semi-colon ';'. This is a useful feature to quickly execute a set of commands and easily recall them using the history cursor keys or !<n>

The ACLI terminal in interactive mode replicates this behaviour across all supported Extreme Networks products. This capability becomes essential in supporting the alias function where it may be desirable for an alias to execute more than one command as well as the command repeat functions when it is desired to repeat not just a single command but a bunch of commands. Simply concatenate the commands using the semi-colon ';' character. ACLI will then execute each command in turn, in scripting mode. The commands will all be executed at the next switch CLI prompt.

```
VSP-8284XSQ:1#% config term; vlan create 666 type port 0; interface vlan 666; ip address 10.0.66.
VSP-8284XSQ:1#% config term
Enter configuration commands, one per line.  End with CNTL/Z.
VSP-8284XSQ:1(config)#% vlan create 666 type port 0
VSP-8284XSQ:1(config)#% interface vlan 666
VSP-8284XSQ:1(config-if)#% ip address 10.0.66.1/24
VSP-8284XSQ:1(config-if)#% end
VSP-8284XSQ:1#%
```

Note that in the above example, only the first line was entered; all the other lines were executed from the first line.

ACLI will do this even on devices which support this feature natively. If it is desired to use the semi-colon ';' natively on the switch, this character will need to be backslahed in ACLI.

Alias Commands

When working on a switch CLI, one typically uses a small bunch of commands 90% of the time. If these commands are long to type every time, this can get frustrating. Even more frustrating, when working on a setup comprising switches of different family types (e.g. VOSS, XOS, ERS...) is that to see the same information one has to enter a slightly different CLI command on each switch type. ACLI's alias capability solves these problems.

ACLI aliases are stored in a text file which is loaded by ACLI on startup. The default alias file is *acli.alias* which is shipped by default with ACLI and contains a number of pre-defined aliases (used by the ACLI author); this file is contained in the ACLI install directory and is a versioned file, which means it is liable to get updated and replaced when the ACLI update script is run, if a newer version of it exists.

If you wish to create your own aliases and do not care to use the pre-defined aliases then you should edit your own *acli.alias* file and place it under one of the following paths:

- ENV path *%ACLI%* (if you defined it)
- ENV path *$HOME/.acli* (on Unix systems)
- ENV path *%USERPROFILE%\.acli* (on Windows)

If instead you are happy to use the pre-defined aliases (and keep obtaining updates for them) and would like to simply define additional personal aliases to merge with the default supplied ones, then you should place your aliases in a file named *merge.alias* and place it in one of the same paths listed above.

The alias file needs to be edited with a specific syntax.

Lines commencing with '#' are comment lines and are ignored

An alias definition begins with a line which starts with the alias command (no spaces or tab prepended). The alias definition can be specified in two ways

1. Simple form, in one line specify the alias name, optional variables, and the command the alias will substitute

```
alias_command [$variable1] [$variable2] ... = <de-reference for alias command>
```

2. Conditional form; in a first line the alias name is specified, with optional variables:

```
alias_command [$variable1] [$variable2] ...
```

Then, on subsequent lines, specify a number of de-reference commands based on certain conditions. These lines must start with space or tab and have 2 fields (separated by '='), in one of these 2 formats:

```
<condition_field> = <de-reference for alias command if condition_field is true>
<condition_field> = &<instruction> [<input based on instruction; can be in double quo
```

The condition field can contain any of Control::CLI:Extreme attributes in {} brackets. You can find available attributes here: https://metacpan.org/pod/Control::CLI::Extreme#Main-I/O-Object-Methods see: attribute() - Return device attribute value.

The condition field can also contain the $variables entered by user after the *alias_command*. The condition field is evaluated as a Perl regular expression, after making the above *{attribute}* & *$variable* replacements. Condition fields are evaluated in order, until one evaluates to true. Once a condition field evaluates to true, the *alias_command* is de-referenced accordingly. If no condition field evaluates to true, then the alias will not resolve.

The variables, if any, defined for the alias can be mandatory or optional. A mandatory variable is a variable which must always be specified with the given alias and is specified by simply adding it after the *alias_command*. An optional variable is a variable which may or may not be appended to the alias and is enclosed in square brackets '[]' when added after the *alias_command*. An alias can be defined with both mandatory and optional variables, on condition that the optional variables come after the mandatory ones.

The de-reference for alias command is the actual command which ACLI will send to the switch if the first form (1) above is used or if the condition_field evaluates to true in the form (2). The optional or mandatory variables can of course be embedded in the command supplied here. If dealing with an optional variable this should again be enclosed in square brackets '[]' which can also include a portion of the final CLI command.

In both the syntaxes above, you can chain multiple commands to send to the switch with semicolons (;) and you can also separate these commands over multiple lines provided that every line begins with one or more space/tab characters and the first non-space character is a semicolon (;) followed by a command.

It is also possible to request alternative actions using the *&<instruction>* format. The following instructions are supported:

- **&print "text to print"** : The alias command will simply print out the text provided; useful for giving syntax
- **&noalias** : This will by bass aliasing all together; useful to avoid clashes with switch commands

On an ACLI session, entering the alias followed by '?' will automatically provide basic syntax for the alias (if it exists). To make a more user friendly syntax for an alias you can also provide a syntax line starting with space or tab then '?:' like this, within the conditional form of the alias definition:

```
?:"Alias syntax:\n     <alias name> [<arg1>] [<arg2>]\n"
```

Or alternatively you display the syntax using the *&print* instruction.

A description of the alias can also be embedded in the definition so that when the alias is listed by the *'@alias list'* command it can be listed with a meaningful description. There are two ways to provide this description field. It can either be provided within the conditional form of the alias definition:

```
~:"<description of what alias does>"
```

Or, for aliases defined using the simple one-liner declaration form it can be included as follows:

```
alias_command [$variable1] [$variable2] ~:"<description>" = <de-reference for alias comma
```

A few examples. All these examples are part of the *acli.alias* file which ships with ACLI.

A simple alias, with less to type.

```
ism = show isis spb-mcast-summary

VSP-8284XSQ:1#% ism
        alias% show isis spb-mcast-summary
================================================================================
                        SPB Multicast - Summary
================================================================================
SCOPE    SOURCE          GROUP           DATA          LSP  HOST
I-SID    ADDRESS         ADDRESS         I-SID    BVID FRAG NAME
--------------------------------------------------------------------------------
There were no entries found.
VSP-8284XSQ:1#%
```

Notice that when an alias is invoked, ACLI will add an echo line immediately after indicating the full command that was replaced for the alias. Alias echoing is by default enabled but can be disabled using the *'@alias echo'* embedded command or the *'alias echo'* command under ACLI control interface.

A simple alias with a mandatory variable:

```
        l2vsn $isid = show isis spbm i-sid all id $isid
```

A simple alias with an optional variable:

```
        dvrdb [$isid] = show dvr database [l3isid $isid]
```

Notice the difference on how the alias is dereferenced with or without the optional argument:

```
VSP8400-1:1#% dvrdb
        alias% show dvr database
================================================================================
                                             DVR DATABASE
================================================================================
                        NEXT                        L3VSN     L2VSN     OUTGOING
DEST          MASK      HOP              VRFID ISID            ISID      INTERFACE
--------------------------------------------------------------------------------
20.0.100.0    255.255.255.0   VSP8400-1      0     0          2800100   cpp
20.0.100.2    255.255.255.255 VSP8400-1      0     0          2800100   cpp
20.0.100.3    255.255.255.255 VSP8400-2      0     0          2800100   3/17
20.0.100.4    255.255.255.255 VSP8400-3      0     0          2800100   3/18
20.0.100.201  255.255.255.255 VSP8400-1      0     0          2800100   X690-1

[...]

20.9.190.0    255.255.255.0   VSP8400-1      9     3800009    2800190   cpp
20.9.190.2    255.255.255.255 VSP8400-1      9     3800009    2800190   cpp
20.9.190.3    255.255.255.255 VSP8400-2      9     3800009    2800190   3/17
20.9.190.4    255.255.255.255 VSP8400-3      9     3800009    2800190   3/18
20.9.190.200  255.255.255.255 VSP7200-2      9     3800009    2800190   2/4

34 out of 34 Total Num of DVR Database entries displayed
acli.pl: Displayed Record Count = 34
--------------------------------------------------------------------------------
VSP8400-1:1#%

VSP8400-1:1#% dvrdb 3800001
        alias% show dvr database l3isid 3800001
================================================================================
                                             DVR DATABASE
================================================================================
                        NEXT                        L3VSN     L2VSN     OUTGOING
DEST          MASK      HOP              VRFID ISID            ISID      INTERFACE
--------------------------------------------------------------------------------
20.1.110.0    255.255.255.0   VSP8400-1      1     3800001    2800110   cpp
20.1.110.2    255.255.255.255 VSP8400-1      1     3800001    2800110   cpp
20.1.110.3    255.255.255.255 VSP8400-2      1     3800001    2800110   3/17
20.1.110.4    255.255.255.255 VSP8400-3      1     3800001    2800110   3/18

4 out of 34 Total Num of DVR Database entries displayed
acli.pl: Displayed Record Count = 4
--------------------------------------------------------------------------------
VSP8400-1:1#%
```

A simple alias which executes more than one CLI command (commands separated with semi-colon ';'):

```
        slamrtp $ip $dscp = config term; application; slamon rtp $ip $dscp; exit; end
```

A conditional alias which shows ethernet ports with link up only:

```
ifup
        {family_type} eq 'PassportERS' && {is_acli}                   = show interfaces
        {family_type} eq 'PassportERS' || {family_type} eq 'Accelar'  = show port inf i
        {family_type} eq 'BaystackERS'                                = show interfaces
        {is_xos}                                                      = show ports info
        {is_isw}                                                      = do show interfa
```

A handy alias to save the config on any switch type:

```
sv [$file]
        {family_type} eq 'PassportERS' || {family_type} eq 'Accelar'    = save config [fi
        {family_type} eq 'BaystackERS' && $file                         = show nvram bloc
        {family_type} eq 'BaystackERS'                                  = copy config nvr
        {family_type} eq 'SecureRouter'                                 = save local [fil
        {family_type} eq 'WLAN9100'                                     = save [$file]
        {is_xos}                                                        = save configurat
        {is_isw} && length($file)                                       = do copy running
        {is_isw}                                                        = do copy running
```

If we check the syntax for any of the alias examples so far, we get to see the alias and its input variables (if any):

```
VSP-8284XSQ:1#% sv ?
        alias% sv [$file]
VSP-8284XSQ:1#%
```

An alias which provides more customized syntax help for the user:

```
tie [$sockname] [$mode]
        ?:"Alias syntax:\n     tie [<socket-name>] [<echo-mode: all|none>]\n"
        $mode eq ''                                     = @socket tie [$sockname]; @socke
        1                                               = @socket tie [$sockname]; @socke
```

Which gives:

```
VSP-8284XSQ:1#% tie ?
Alias syntax:
        tie [<socket-name>] [<echo-mode: all|none>]

VSP-8284XSQ:1#%
```

A more advanced alias which uses semi-colon separated commands over multiple lines and a different way to show the alias syntax if executed with no argument:

```
ersupl [$upl]
        $upl eq ''      = &print "Alias syntax:\n     ersupl \n"
        1               = fa extended-logging
                        ; vlan configcontrol automatic
                        ; mlt 1 enable member $upl learning disable
                        ; mlt 1 loadbalance advance
                        ; vlacp macaddress 01:80:c2:00:00:0f
                        ; interface Ethernet $upl
                        ;  vlacp timeout short
                        ;  vlacp timeout-scale 5
                        ;  vlacp enable
                        ; exit
                        ; vlacp enable
```

Finally an alias which makes use of the *&noalias* instruction:

```
tgz [$delete]
        !{is_voss}                      = &noalias
        $delete eq '?'                  = &print "Alias syntax:\n     tgz [delete]\n"
        $delete eq 'delete'             = delete *.tgz -y
        $delete eq ''                   = ls *.tgz
```

If the above alias is executed on a session with a device which is not a VOSS switch, the *tgz* command is not deemed an alias (and the *tgz* command will be sent to the connected host switch as is).

You may also refer to the *acli.alias* file shipped with ACLI, which contains comments on its syntax and plenty of other examples to play with.

Note: Care should be taken to choose *alias_names* which do not conflict with a switch command. If there is a conflict, the alias command will override the switch command. If this happens, it is possible to force a command to the switch (without de-aliasing) by appending ';' to the command. Alternatively include a condition match to avoid the clash using the *&noalias* instruction.

The alias functionality can be disabled/enabled using the following command:

```
@alias disable|enable
```

The alias files, *acli.alias* + *merge.alias* (if it exists) are always loaded when ACLI is launched. You can edit the alias files while ACLI sessions are running, but each ACLI session will keep running with the aliases that it loaded on startup. To reload the alias files on already running ACLI sessions, you can use the embedded *'@alias reload'* command or the *'alias reload'* command under the ACLI control interface. The same *'@alias'* embedded command and *'alias'* command under ACLI control interface also offer commands to show the loaded aliases.

```
VSP-8284XSQ:1#% @alias ?
Syntax: @alias disable|echo|enable|info|list|reload|show

VSP-8284XSQ:1#% @alias reload
Loading alias file: C:\Users\lstevens\Scripts\acli\acli.alias
Merging alias file: C:\Users\lstevens\.acli\merge.alias
Successfully re-loaded default & merge alias files

VSP-8284XSQ:1#%
```

Available alias commands can be listed using either of these embedded commands:

```
@alias list [<description-search-pattern>]
@alias show [<pattern>]
```

The *'@alias list'* will produce a compact list of all available aliases with a description of what they do. A search pattern can be provided on the command itself or one can also use ACLI's regular grep capability.

```
VSP-8284XSQ:1#% @alias list spb

dropstat [$ports]             : Dump SPB ISIS drop-stats (ERS8k/VSP9k only)
dvr                           : Show SPB DVR global info
dvrbh [$isid]                 : Show SPB DVR backbone
dvrbm                         : Show SPB DVR backbone members
dvrdb [$isid]                 : Show SPB DVR database
dvrh [$vrf]                   : Show SPB DVR hosts
dvri [$vrf]                   : Show SPB DVR interfaces
dvrm                          : Show SPB DVR domain members
dvrr [$vrf]                   : Show SPB DVR routes
fibi6                         : Dump SPB ISIS IPv6 routes installed
fibip [$isid]                 : Dump SPB ISIS IP routes installed
fibm [$arg1] [$arg2] [$arg3]  : Dump SPB's multicast forwarding database (fdb)
fibu [$arg1] [$arg2]          : Dump SPB's unicast forwarding database (fdb)
isa                           : Show SPB ISIS interface authentication
isdb                          : Show SPB ISIS LSDB
isi                           : Show SPB ISIS interfaces
isj                           : Show SPB ISIS adjacencies
ism                           : Show SPB fabric known IP Multicast sources
ismd [$vlanvrf]               : Show SPB fabric IP Multicast streams for given VSN
isname [$id] [$name]          : Re-configure SPB system name without having to modify
nick                          : Show all SPB nodes in fabric
nni $ports [$speed]           : Configure SPB NNI ports and set metric according to po
nnimetric $ports $speed       : Modify SPB NNI ports metric according to port speed
spb                           : Show SPB global info
spgf [$vrf]                   : Show SPB PIM Gateway PIM side sources for given VRF or
spgi [$vrf]                   : Show SPB PIM Gateway interfaces for given VRF or GRT
spgn [$vrf]                   : Show SPB PIM Gateway neighbours for given VRF or GRT
spgr [$vrf]                   : Show SPB PIM Gateway multicast routes for given VRF or
spgs [$vrf]                   : Show SPB PIM Gateway fabric side sources for given VRF

VSP-8284XSQ:1#%
```

The *'@alias show'* command will give a more detailed description of the what the evaluation logic of the alias is.

```
VSP-8284XSQ:1#% @alias show ifup

ifup
    IF {family_type} eq 'PassportERS' && {is_acli}
```

```
        THEN:
            show interfaces gigabitEthernet interface ||up\s+up

    IF {family_type} eq 'PassportERS' || {family_type} eq 'Accelar'
        THEN:
            show port inf interface ||up\s+up

    IF {family_type} eq 'BaystackERS'
        THEN:
            show interfaces ||up

    IF {is_xos}
        THEN:
            show ports information ||active

    IF {is_isw}
        THEN:
            do show interface * status !!Down


VSP-8284XSQ:1#%
```

Variables

Variables are another major feature of the ACLI terminal and are also only available in interactive mode. There are many ways to use variables. They come in handy when having to deal with lists of port numbers, which would otherwise be painful to have to type in every time. The ACLI terminal is particularly geared towards easily capturing port lists or ranges into variables. Variables also come in handy when tie-ing multiple terminals together via the socket functionality, as it allows the user to perform configuration on one terminal which gets executed across many ACLI sessions, and since usually the port numbers and other IDs are not the same across the different switches, these differences can be stored in appropriate variables which when referred to in the driving terminal will be dereferenced to the appropriate value for each and every ACLI session. Once a value is stored in a variable, the variable can be dereferenced by simply embedding it in CLI commands or eval-ed in Perl code also embedded in CLI commands, and they can be used in ACLI's scripting conditional operators as well.

All ACLI perl variables are pre-pended with the dollar '$' sign (much like Perl's scalar variables). Multiple variable types exist.

User variables

User variables are variables defined by the user in the format *$<var-name>*. The variable name can be made of one or more alphanumeric characters where word characters are case sensitive. The underscore '_' character is allowed in variable names, but not as the 1st character following the dollar sign, with the exception of the variable *$_* which is a special variable called the default variable. The minus sign '-' is not allowed in variable names.

A user variable comes into existence the moment a value is assigned to it, either via explicit assignment or via capturing.

Default variable

The default variable is named *$_* but can also be referred to as simply *$* with no name characters following it. It behaves like any other user variable and its intent is simply to be used as a convenience temporary variable to store values which you do not intend to hold on to for long. It is convenient also in that it is quick to type as it can be called with a single character '$'. However, if using the default variable in ACLI scripting conditional operators (@if, @while, @until, etc..) then it must always be referred to as *$_*. Unlike other user variables, the default variable, if set, cannot be saved with the '@save' command.

Setting user variables

A user variable can be set simply by making an assignment to it.

```
VSP-8284XSQ:1#% $myvar = 1/1

$myvar      = 1/1

VSP-8284XSQ:1#%
```

Any string or number can be assigned to a variable.

```
VSP-8284XSQ:1#% $ = blah blah

$_          = blah blah

VSP-8284XSQ:1#%
```

The default variable was used in the above example. If assigning text to a variable, leading and trailing spaces are removed. If you wanted to preserve leading or trailing spaces then enclose the string in single or double quotes

```
VSP-8284XSQ:1#% $text = "   indented blah"

$text       = '   indented blah'

VSP-8284XSQ:1#%
```

To overwrite a variable with a different value, simply assign a new value to the same variable.

To append a new value to an existing variable, there are a couple of ways of doing this:

```
VSP-8284XSQ:1#% $text = "$text even more blah"
          vars% $text = "   indented blah even more blah"

$text       = '   indented blah even more blah'

VSP-8284XSQ:1#%
```

In the above example, we are assigning a new string to $text, but part of that string is what $text was already set with. Notice that you need to use double quotes here (as variables are not dereferenced inside single quotes). Or else we do it without quotes, if we don't care about leading/trailing spaces:

```
VSP-8284XSQ:1#% $text = $text, enough
          vars% $text =    indented blah even more blah, enough

$text       = indented blah even more blah, enough

VSP-8284XSQ:1#%
```

Or we could have used the '.=' append operator to achieve the same:

```
VSP-8284XSQ:1#% $text .= really enough

$text       = indented blah even more blah, enough,really enough

VSP-8284XSQ:1#%
```

Note that ACLI always stores all variables as just strings of text. However, if there are commas in there, then the variable will chopped up into a list when passed to the ACLI repeat operator '&' or the ACLI scripting '@for' loop. Use of the '.=' operator automatically includes a comma when appending the new value to existing values (if the variable was already set).

This makes more sense when dealing with port numbers; so if we wanted to add another port to our $myvar variable:

```
VSP-8284XSQ:1#% $myvar .= 1/5

$myvar      = 1/1,1/5

VSP-8284XSQ:1#%
```

Whereas to delete a variable, simply assign nothing to it:

```
VSP-8284XSQ:1#% $text=
VSP-8284XSQ:1#%
VSP-8284XSQ:1#% @vars show $text
No variables found matching $text

VSP-8284XSQ:1#%
```

User variables are particularly geared towards numbers and port numbers. These can be entered as lists or port ranges and are automatically converted into comma separated lists, but by default displayed as ranges. This might sound a bit confusing, let's go through some examples.

```
VSP-8284XSQ:1#% $range = 1-10

$range      = 1-10

VSP-8284XSQ:1#% @vars raw $range

$range      = 1,2,3,4,5,6,7,8,9,10

VSP-8284XSQ:1#% @vars show $range

$range      = 1-10

VSP-8284XSQ:1#%
```

We set $range to 1-10. Visibly the variable is set to what we set it at. The '@vars raw' command displays variables as they are actually stored internally in ACLI, i.e. as a comma separated list. Whereas the '@vars show' command will always display more compact ranges, where possible. When using this variable in a CLI command, what will be used is what is shown when simply recalling the variable (i.e. a range in this case). To force ACLI to dereference the raw value instead you can use the $' notation for the same variable:

```
VSP-8284XSQ:1#% @print $range
        vars% @print 1-10
1-10

VSP-8284XSQ:1#% @print $'range
        vars% @print 1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10

VSP-8284XSQ:1#%
```

If we wanted to create VLANs or MLTs 1-10 on VOSS, we know there is no single VOSS command which will do that, so we would have no other choice than to use ACLI's repeat operator to achieve this:

```
VSP-8284XSQ:1(config)#% mlt %s &'$range
               vars% mlt %s &'1-10
VSP-8284XSQ:1(config)#% mlt 1
VSP-8284XSQ:1(config)#% mlt 2
VSP-8284XSQ:1(config)#% mlt 3
VSP-8284XSQ:1(config)#% mlt 4
VSP-8284XSQ:1(config)#% mlt 5
VSP-8284XSQ:1(config)#% mlt 6
VSP-8284XSQ:1(config)#% mlt 7
VSP-8284XSQ:1(config)#% mlt 8
VSP-8284XSQ:1(config)#% mlt 9
```

```
VSP-8284XSQ:1(config)#% mlt 10
VSP-8284XSQ:1(config)#%
```

Note that the ACLI repeat operator '&' does not automatically expand ranges; we need to add the ' character to force it to expand ranges with "&'". Note that we could have achieved the same result by forcing the variable to dereference its raw value like this: *mlt %s &$'range*. The same is true with the *@for* operator.

Or we could use ACLI scripting to do the same (deleting the MLTs in this case..); copy-pasting this ACLI script:

```
@for $_  &'$range
         no mlt $
@endfor
```

Gives these commands:

```
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 1
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 2
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 3
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 4
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 5
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 6
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 7
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 8
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 9
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#% @for $_  &$range
VSP-8284XSQ:1(config)#%   no mlt $
                 vars%  no mlt 10
VSP-8284XSQ:1(config)#% @endfor
VSP-8284XSQ:1(config)#%
```

In the case of port numbers, ranges are expanded according to the knowledge ACLI has of the port layout of the connected device.

```
VSP-8284XSQ:1#% $portRange = 1/1-10/10

$portRange    = 1/1-1/42,2/1-2/42

VSP-8284XSQ:1#%
VSP-8284XSQ:1#% @vars raw $portRange

$portRange    = 1/1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9,1/10,1/11,1/12,1/13,1/14,1/15,1/16,1/
```

```
VSP-8284XSQ:1#%
```

In the above example, ACLI was connected to a VSP8200, which has 42 ports per slot and 2 slots in total. Hence the port range assigned to our $portRange was pruned back to match the available ports on the connected device. Again '@vars raw' command shows that the variable is internally storing the port range as a list. Whereas for display purposes and when dereferencing the variable, a more compact port range will be used. ACLI will default to compacting ports into ranges which do not span slots, though this is configurable though using the *'terminal portrange'* command under ACLI control interface or via the *default_port_range_mode* key in the *acli.ini* file.

The way port ranges are rendered in slot/port format also depend on the device type ACLI is connected to. If we define a port range where all ports are on the same slot on a VOSS VSP we will get this:

```
VSP-8284XSQ:1#% $portRange = 1/1-10

$portRange  = 1/1-1/10

VSP-8284XSQ:1#%
```

If we make the same variable assignment on an ERS stack, we will get a different range format:

```
ERS4900-STK#% $portRange = 1/1-10

$portRange  = 1/1-10

ERS4900-STK#%
```

Notice that ACLI variable is now compacting the port range as *x/1-50* and not as *x/1-x/50* as was the case with VOSS.

The same is true if we make that assignment on an XOS stack:

```
Slot-1 X460G2-STK.9 #% $portRange = 1:1-10

$portRange  = 1:1-10

Slot-1 X460G2-STK.9 #%
```

ExtremeXOS stacks support the same type of port ranges as ERS, and also use the colon ':' character instead of the more common slash '/' for denoting slot/port numbers.

In general, when assigning to variables, ACLI will take port ranges in any valid input format, but will always display and de-reference the variable using only the range format supported by the connected device.

So, for example, *ALL* is a valid ERS port range, hence we have:

```
ERS4900-STK#% $allPorts = all

$allPorts    = 1/1-50,2/1-50,3/1-50

ERS4900-STK#% @vars raw $allPorts

$allPorts    = 1/1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9,1/10,1/11,1/12,1/13,1/14,1/15,1/16,1/

ERS4900-STK#%
```

Likewise on a VOSS VSP:

```
VSP-8284XSQ:1#% $allPorts = all

$allPorts    = 1/1-1/42,2/1-2/42

VSP-8284XSQ:1#% @vars raw $allPorts

$allPorts    = 1/1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9,1/10,1/11,1/12,1/13,1/14,1/15,1/16,1/
```

```
VSP-8284XSQ:1#%
```

And on our same XOS:

```
Slot-1 X460G2-STK.9 #% $allPorts = all

$allPorts    = 1/1-54,2/1-54,3/1-34

Slot-1 X460G2-STK.9 #% @vars raw $allPorts

$allPorts    = 1/1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9,1/10,1/11,1/12,1/13,1/14,1/15,1/16,1/

Slot-1 X460G2-STK.9 #%
```

Note, if you wanted the variable to hold the string *"all"* instead, then simply put quotes around it.

Basically ACLI's port ranges will always adapt based on what is supported by the connected device. This is necessary, because we want to be able to dereference our variables directly into CLI commands for the device we are connected to. If ACLI were to use the wrong format of port range then the device would simply throw a syntax error on the CLI command and it would not be accepted.

In the case of a connected device which does not support port ranges at all (this was the case with the historical SecureRouter) then ACLI would not use any port ranges to display variables (author no longer has any SecureRouters for screenshots!) and would use a list format instead.

Two variable modifiers exist. The first one using the *$#* notation allows the variable to dereference not its value but the number of comma separated elements it holds. The second using the *$'* notation forces ACLI to dereference the variable in raw mode without attempting to compact the value in ranges

```
VSP-8284XSQ:1#% $allPorts

$allPorts    = 1/1-1/42,2/1-2/42

VSP-8284XSQ:1#% $#allPorts

$#allPorts   = 84

VSP-8284XSQ:1#% $'allPorts

$allPorts    = 1/1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9,1/10,1/11,1/12,1/13,1/14,1/15,1/16,1/

VSP-8284XSQ:1#%
```

Capturing user variables

So we have seen that it is fairly easy to assign values to variables. However a much more compelling way to set the variables is to capture them directly from the output of CLI show commands. This is particularly useful when driving several ACLI sessions with the socket feature as it allows each ACLI session to correctly populate the same variable with device specific values.

To capture ports to a variable simply use the redirecton symbol '>' or '>>' followed by the destination variable.

```
CLI show command > $variable [-g]
CLI show command >> $variable [-g]
```

If the variable was already set, use of '>' will result in the variable being overwritten with the newly captured values; whereas use of '>>' will result in the new captured values being comma appended to the existing values already held in the variable. The optional -g modifier allows capturing of all ports seen in every line of output, as opposed to only the first occurrence in each line of output.

Note, the redirect symbols '>'/'>>' followed by a variable does variable capturing; whereas the same redirect symbols '>'/'>>' followed by a file name does output redirection.

Variable capturing in its simplest form is only tuned for capturing port numbers in the output of CLI show commands. For example, if we wanted to capture all the SPB NNI ports on a VOSS switch we could simply use this:

```
VSP8400-1:1#% isi > $nni
      alias% show isis interface > $nni
================================================================================
                             ISIS Interfaces
================================================================================
IFIDX           TYPE     LEVEL    OP-STATE  ADM-STATE  ADJ   UP-ADJ  SPBM-L1-METRIC
--------------------------------------------------------------------------------
Port2/1         pt-pt    Level 1  UP        UP         1     1       50
Port2/2         pt-pt    Level 1  UP        UP         1     1       50
Port2/3         pt-pt    Level 1  UP        UP         1     1       50
Port2/4         pt-pt    Level 1  UP        UP         1     1       50
Port3/17        pt-pt    Level 1  UP        UP         1     1       50
Port3/18        pt-pt    Level 1  UP        UP         1     1       50


--------------------------------------------------------------------------------
 6 out of 6 Total Num of ISIS interfaces
acli-dev.pl: Displayed Record Count = 6
--------------------------------------------------------------------------------

Var $nni = 2/1-2/4,3/17-3/18

VSP8400-1:1#%
```

Or if we wanted simply to capture all ports with link up on the switch, we could do this:

```
VSP8400-1:1#% ifup > $up
      alias% show interfaces gigabitEthernet interface ||up\s+up > $up
================================================================================
                             Port Interface
================================================================================
PORT                          LINK  PORT           PHYSICAL          STATUS
NUM      INDEX DESCRIPTION     TRAP  LOCK    MTU    ADDRESS           ADMIN  OPERATE
--------------------------------------------------------------------------------
2/1      256   40GbCR4         true  false   9600   b0:ad:aa:4f:0c:20 up     up
2/2      260   40GbCR4         true  false   9600   b0:ad:aa:4f:0c:24 up     up
2/3      264   40GbCR4         true  false   9600   b0:ad:aa:4f:0c:28 up     up
2/4      268   40GbCR4         true  false   9600   b0:ad:aa:4f:0c:2c up     up
2/5      272   40GbCR4         true  false   9600   b0:ad:aa:4f:0c:30 up     up
3/17     336   40GbCR4         true  false   9600   b0:ad:aa:4f:0c:50 up     up
3/18     340   40GbCR4         true  false   9600   b0:ad:aa:4f:0c:54 up     up
```

```
                    Var $up = 2/1-2/5,3/17-3/18

                    VSP8400-1:1#%
```

In general, we simply execute the show command we need, perhaps using grep to filter out only the ports we are interested in and then we capture those ports into a variable. So if we wanted to capture a variable with just the 40GbE ports on the switch we could do this:

```
VSP8400-3:1#% if || 40G > $fast
        alias% show interfaces gigabitEthernet interface!!locked || 40G > $fast
================================================================================
                              Port Interface
================================================================================
PORT                          LINK  PORT        PHYSICAL           STATUS
NUM     INDEX DESCRIPTION     TRAP  LOCK   MTU  ADDRESS            ADMIN OPERATE
--------------------------------------------------------------------------------
1/17    208   40GbNone        true  false  9600 64:6a:52:c5:5c:10 down  down
1/18    212   40GbNone        true  false  9600 64:6a:52:c5:5c:14 down  down
2/1     256   40GbCR4         true  false  9600 64:6a:52:c5:5c:20 up    up
2/2     260   40GbCR4         true  false  9600 64:6a:52:c5:5c:24 up    up
2/3     264   40GbNone        true  false  9600 64:6a:52:c5:5c:28 down  down
2/4     268   40GbNone        true  false  9600 64:6a:52:c5:5c:2c down  down
2/5     272   40GbSR4         true  false  9600 64:6a:52:c5:5c:30 up    up
2/6     276   40GbNone        true  false  9600 64:6a:52:c5:5c:34 down  down
3/17    336   40GbNone        true  false  9600 64:6a:52:c5:5c:50 down  down
3/18    340   40GbCR4         true  false  9600 64:6a:52:c5:5c:54 up    up

Var $fast = 1/17-1/18,2/1-2/6,3/17-3/18

VSP8400-3:1#%
```

And if we wanted to append 100GbE ports to the same variable...

```
VSP8400-3:1#% if || 100G >> $fast
        alias% show interfaces gigabitEthernet interface!!locked || 100G >> $fast
================================================================================
                              Port Interface
================================================================================
PORT                          LINK  PORT        PHYSICAL           STATUS
NUM     INDEX DESCRIPTION     TRAP  LOCK   MTU  ADDRESS            ADMIN OPERATE
--------------------------------------------------------------------------------
4/1     384   100GbCR4        true  false  9600 64:6a:52:c5:5c:60 up    up
4/2     385   100GbCR4        true  false  9600 64:6a:52:c5:5c:61 up    up

Var $fast = 1/17-1/18,2/1-2/6,3/17-3/18,4/1-4/2

VSP8400-3:1#%
```

Configuring these ports is now simply a matter of doing:

```
VSP8400-3:1#% conf
Configuring from terminal or network [terminal]?
Enter configuration commands, one per line.  End with CNTL/Z.
VSP8400-3:1(config)#% ife $fast
            vars% ife 1/17-1/18,2/1-2/6,3/17-3/18,4/1-4/2
            alias% interface gigabitEthernet 1/17-1/18,2/1-2/6,3/17-3/18,4/1-4/2
VSP8400-3:1(config-if)#%
```

Capturing of port numbers will always extract the first occurrence of port numbers/list/range from each line of output. In the following example, only the port numbers in column 1 are captured and assigned to the *$p* variable:

```
VSP8200-1:1#% tdp > $p
        alias% show autotopology nmm-table !0/ *0 > $p
================================================================================
                              Topology Table
================================================================================
Local                                                                       Rem
Port      IpAddress        SegmentId MacAddress   ChassisType        BT LS CS Port
```

```
        ----------------------------------------------------------------------------------
        1/3       20.0.209.4      0x000131  b42d5653cc01 ERS4950GTS-PWR+         12 Yes HtBt  1/49
        1/41      20.0.0.11       0x000305  f873a20780d0 VSP8608                 12 Yes HtBt  3/5
        1/42      20.0.0.12       0x000305  f873a203a0d0 VSP8608                 12 Yes HtBt  3/5
        2/1       20.0.209.11     0x000131  b42d56556400 ERS4950GTS-PWR+         12 Yes HtBt  1/49
        2/2       20.0.209.12     0x000119  d4785607fc00 ERS5928GTS-UPWR         12 Yes HtBt  1/25
        2/3       20.0.209.13     0x000119  506184fbd000 ERS4826GTS-PWR+         12 Yes HtBt  1/25
        2/41      20.0.20.32      0x000229  b0adaa429468 VSP8284XSQ              12 Yes HtBt  2/41

        $p         = 1/3,1/41-1/42,2/1-2/3,2/41

        VSP8200-1:1#%
```

In older versions of ACLI, up to version 4.01, this was not the case and all ports in the above output would have been captured, including the port numbers listed in the last column. This was not hugely useful, as there is little use in mashing up the local and remote port numbers from the example above, so this behaviour has changed in ACLI 4.02. There could however be other cases where it might be desireable to do so, and so to obtain the old behaviour a *-g* modifier can be applied to variable port capturing, which if applied to the above example allows us to capture all ports anyway:

```
        VSP8200-1:1#% tdp > $p -g
             alias% show autotopology nmm-table !0/ *0 > $p -g
        ================================================================================
                                      Topology Table
        ================================================================================
                                                                                    Rem
        Local
        Port      IpAddress       SegmentId MacAddress   ChassisType            BT LS CS  Port
        --------------------------------------------------------------------------------
        1/3       20.0.209.4      0x000131  b42d5653cc01 ERS4950GTS-PWR+         12 Yes HtBt  1/49
        1/41      20.0.0.11       0x000305  f873a20780d0 VSP8608                 12 Yes HtBt  3/5
        1/42      20.0.0.12       0x000305  f873a203a0d0 VSP8608                 12 Yes HtBt  3/5
        2/1       20.0.209.11     0x000131  b42d56556400 ERS4950GTS-PWR+         12 Yes HtBt  1/49
        2/2       20.0.209.12     0x000119  d4785607fc00 ERS5928GTS-UPWR         12 Yes HtBt  1/25
        2/3       20.0.209.13     0x000119  506184fbd000 ERS4826GTS-PWR+         12 Yes HtBt  1/25
        2/41      20.0.20.32      0x000229  b0adaa429468 VSP8284XSQ              12 Yes HtBt  2/41

        $p         = 1/3,1/25,1/41-1/42,1/49,2/1-2/3,2/41,3/5

        VSP8200-1:1#%
```

Capturing ports into variables as shown above only works on devices which use a slot/port, or slot/port/channel or slot:port notation. So none of this will work on an ERS or an XOS standalone switch, since these number ports with a single decimal value, usually between 1-50. Given that switch CLI commands are full of numbers between 1-50 which are not referring to port numbers it would not be safe to try and capture these values into variables.

But there are a couple of other ways we can use to capture values to variables. These require the user to either specify which column of the output data to capture from or else to provide a regular expression to specify what and where to capture.

The first form is to follow the variable name with a '%' character immediately followed by the column number where we want to capture data. Columns are separated by one or more space characters.

```
        CLI show command > $variable %<n>
```

This form is best used in conjunction with simple grep '|' in such a way to only display (and hence capture from) the lines where we have data to capture from, and eliminating the command's show banners which would otherwise become part of the capture data. Here below we capture the SPB NNI ports on an ERS standalone switch.

```
        ERS5900-FC#% isi
             alias% show isis interface
        ================================================================================
                                      ISIS Interfaces
        ================================================================================
        IFIDX         TYPE    LEVEL    OP-STATE  ADM-STATE  ADJ UP-ADJ SPBM-L1-METRIC
```

```
================================================================================
Port: 27        pt-pt   Level 1  UP          UP          1   1      200
Port: 28        pt-pt   Level 1  UP          UP          1   1      200
ERS5900-FC#%
ERS5900-FC#% isi |Port: > $nni %2
     alias% show isis interface |Port: > $nni %2
Port: 27        pt-pt   Level 1  UP          UP          1   1      200
Port: 28        pt-pt   Level 1  UP          UP          1   1      200

Var $nni = 27-28

ERS5900-FC#%
```

This method in fact becomes a very efficient method for extracting any data from CLI show comands. The simple grep selects the rows containing the data and the variable capture %<n> determines the columns to read.

```
VSP8400-3:1#% show snmp-server user
Engine ID = 80:00:08:E0:03:64:6A:52:C5:5C:00

==========================================================================================
                            USM Configuration
==========================================================================================
User/Security Name    Engine Id                        Protocol
------------------------------------------------------------------------------------------
admin              0x80:00:08:E0:03:64:6A:52:C5:5C:00 HMAC_SHA, NO  PRIVACY
adminaes           0x80:00:08:E0:03:64:6A:52:C5:5C:00 HMAC_SHA, AES PRIVACY,
operator           0x80:00:08:E0:03:64:6A:52:C5:5C:00 HMAC_SHA, NO  PRIVACY

3 out of 3 Total entries displayed
acli.pl: Displayed Record Count = 4
----------------------------------------------------------------------------------
VSP8400-3:1#%
VSP8400-3:1#% show snmp-server user |0x > $users %1
admin              0x80:00:08:E0:03:64:6A:52:C5:5C:00 HMAC_SHA, NO  PRIVACY
adminaes           0x80:00:08:E0:03:64:6A:52:C5:5C:00 HMAC_SHA, AES PRIVACY,
operator           0x80:00:08:E0:03:64:6A:52:C5:5C:00 HMAC_SHA, NO  PRIVACY

Var $users = admin,adminaes,operator

VSP8400-3:1#%
```

The same variable capture syntax can also be used to capture the data in two or more columns into the same variable, using these syntaxes:

```
CLI show command > $variable %<n1>[%<n2>]...
CLI show command > $variable %<n1>-[%<n3>]...
```

The '%' cloumn indicators can be concatenated, e.g. '%1%3', or comma separated, '%1,%3'. Ranges are also allowed, like '%1-%5', as well an unbounded ranges, like '%5-', which will result in the data in all columns from column 5 onwards to be captured. A few examples:

```
VSP8600-1:1#% show license |License granted for slots > $slots %6%7%8
        License granted for slots       : 1  2  3  4  5  6  7  8

Var $slots = 1-3

VSP8600-1:1#%

VSP8600-1:1#% show license |License granted for slots > $slots %6-%13
        License granted for slots       : 1  2  3  4  5  6  7  8

Var $slots = 1-8

VSP8600-1:1#%

VSP8600-1:1#% show license |License granted for slots > $slots %6-
        License granted for slots       : 1  2  3  4  5  6  7  8
```

```
            Var $slots = 1-8

            VSP8600-1:1#%
```

Another variant of the same syntax allows multiple variables to be captured in the same CLI command.

```
        CLI show command > $variable1,$variable2,... %<n1>%<n2>...
        CLI show command > $variable1,$variable2,$variable3... %<n1>-%<n3>
```

Again, the '%' cloumn indicators can be concatenated, e.g. '%1%3', or comma separated, '%1,%3' or defined as ranges, '%1-%5'. Though unbounded ranges are not allowed here. In general with this syntax ACLI will check to make sure that the number of '%' columns requested in the variable capture command matches the number of capture variables provided. If the number is not the same, then the command is not accepted with an error. An example:

```
        VSP8400-3:1#% lldn
              alias% show lldp neighbor summary
        ====================================================================================
                                                    LLDP Neighbor Summary
        ====================================================================================
        LOCAL          IP            CHASSIS          REMOTE
        PORT     PROT  ADDR          ID               PORT            SYSNAME        SYSD
        ------------------------------------------------------------------------------------
        2/1      LLDP  20.0.10.73    a4:25:1b:52:24:00  2/1                           VSP7200-3      VSP-
        2/2      LLDP  20.0.10.74    a4:78:86:fb:e0:00  2/1                           VSP7200-4      VSP-
        2/5      LLDP  20.0.109.12   00:04:96:a5:12:72  49                            X690-2         Extr
        3/18     LLDP  20.0.10.21    b0:ad:aa:4f:0c:00  3/18                          VSP8400-1      VSP-
        4/1      LLDP  20.0.0.11     f8:73:a2:07:80:00  1/3                           VSP8600-1      VSP-
        4/2      LLDP  20.0.0.12     f8:73:a2:03:a0:00  1/3                           VSP8600-2      VSP-
        ------------------------------------------------------------------------------------
        Total Neighbors : 6
        VSP8400-3:1#% lldn | LLDP  \d > $locPort,$remport %1,%5
              alias% show lldp neighbor summary | LLDP > $locPort,$remport %1,%5
        2/1      LLDP  20.0.10.73    a4:25:1b:52:24:00  2/1                           VSP7200-3      VSP-
        2/2      LLDP  20.0.10.74    a4:78:86:fb:e0:00  2/1                           VSP7200-4      VSP-
        2/5      LLDP  20.0.109.12   00:04:96:a5:12:72  49                            X690-2         Extr
        3/18     LLDP  20.0.10.21    b0:ad:aa:4f:0c:00  3/18                          VSP8400-1      VSP-
        4/1      LLDP  20.0.0.11     f8:73:a2:07:80:00  1/3                           VSP8600-1      VSP-
        4/2      LLDP  20.0.0.12     f8:73:a2:03:a0:00  1/3                           VSP8600-2      VSP-

        Var $locPort = 2/1-2/2,2/5,3/18,4/1-4/2
        Var $remport = 1/3,2/1,3/18,49

        VSP8400-3:1#%
```

The other form to do variable capturing is to use regular expressions with grouping brackets '()'. The syntax is the following:

```
        CLI show command > $variable1,$variable2,... '<regex>'[ig]
        CLI show command > $variable1,$variable2,... '<regex>' [-ig]
```

Where '*<regex>*' is the regular expression which must include as many grouping brackets '()' as there are variables to capture. The regex should be provided inside single quotes and the closing quote can be optionally followed by a lower case 'i', which will make the regex patterns case insensitive, or a lower case 'g' which will allow the regex to match multiple times per line of output. Both the 'i' and 'g' modifiers can also be provided as *-i* or *-g* following the regex.

This form is useful when the data to capture is not nicely delimited by spaces. Follows an example:

```
        Slot-1 X460G2-STK.8 #% show log configuration |'syslog;' > $ip,$port '(\d+\.\d+\.\d+\.\d+
        Log Target    : syslog; 10.8.255.15:514 (vr VR-Mgmt), local0

        $ip         = 10.8.255.15
        $port       = 514
```

```
        Slot-1 X460G2-STK.9 #%
```

In general, the same form can be used if one wishes to capture output for separate variables even if the values appear on separate lines. In the following example, we capture all the L2VSN and L3VSN I-SIDs used by a particular fabric node by dumping the ISIS LSDB; the list of ISIDs is to be stored in two separate variables, for L2 and L3, but the values we intend to capture will appear on different output lines (i.e. we will not get a both a L3 I-SID and a L2 I-SID value on the same line of output); so the provided *regex* will need to provide alternate patterns separated by the '|' character. In addition to that, there are multiple L2 I-SID values listed on the same line of output, so we need our capturing regular expression to capture all occurrencies on each line (not just the first one) so we also add the *g* flag to the capture regex:

```
VSP8400-1:1#% show isis lsdb sysid 82bb.0000.2100 detail |Both,Rx,Tx,Vrf ISID: -s > $l2is
                        16000000(Rx),16777215(None)
                        2800100(Rx),2800101(Rx),2800104(Rx),2800109(Rx),2800110(Rx),28001
                        2800190(Rx),2800191(Rx),2801111(Rx),10002122(Rx)
                        2800100(Both),2800101(Both),2800104(Both),2800109(Both),2800110(B
                        2800190(Both),2800191(Both),2801111(Both),10002122(Both),16678216
            Vrf ISID:3800001
            Vrf ISID:3800002
            Vrf ISID:3800003
            Vrf ISID:3800009


$l2isid    = 2800100-2800101,2800104,2800109-2800111,2800120,2800130,2800190-2800191,280
$l3isid    = 3800001-3800003,3800009

VSP8400-1:1#%
```

The above example is also used by the pre-defined *'dbisid'* alias.

Advanced user variables

The above sections have so far only covered ACLI simple (scalar) variables, wihch for most uses are sufficient. Yet for storing data in a structured order or one value in relation to another this is only possible with arrays or hashes. So ACLI supports both arrays and hashes as well.

Array variables are denoted by simply appending '[]' to a regular user variable name:

```
$list[]
```

Hash variables are denoted by appending '{}':

```
$hash{}
```

Note that ACLI only supports one variable namespace, so a given variable name can have values assigned either as a list or hash or a scalar. Hence assigning a value to *$list* would erase the above array and relace it with a scalar.

Setting array variables

In practice array variables will typically get set by capturing a list of values from the output of some CLI command, and this will be covered in the section below. Nevertheless, a syntax is available to manually assign a list of values to an array variable.

```
VSP-8284XSQ:1#% $list[] = (1/1-1/10; 1,2,3; string)

$list[]     = (1/1-1/10; 1-3; string)

VSP-8284XSQ:1#%
```

Each element of the array can take the same values and follows the same rules as scalar variables which have been covered in the preceding sections.

Note that ACLI arrays are 1-based so 1 is the index for the first element in the array. The 0 index is allowed but will return the last element in the array. Negative index values are not allowed.

```
VSP-8284XSQ:1#% $list[1]

$list[1]     = 1/1-1/10

VSP-8284XSQ:1#% $list[2]

$list[2]     = 1-3

VSP-8284XSQ:1#% $list[3]

$list[3]     = string

VSP-8284XSQ:1#% $list[0]

$list[3]     = string

VSP-8284XSQ:1#%
```

Array values can also be directly assigned to an index value, as long as the index provided is already within the array or it increases the array size by 1.

```
VSP-8284XSQ:1#% $list[3] = newstring

$list[3]     = newstring

VSP-8284XSQ:1#% $list[4] = 99
```

```
$list[4]    = 99

VSP-8284XSQ:1#% $list

$list[]    = (1/1-1/10; 1-3; newstring; 99)

VSP-8284XSQ:1#%
```

Note from the above example that we have replaced the value of the 3rd and last element and we have appended a new element increasing the size of the array by 1. This allows us to grow arrays in iteration loops while preventing a user from creating an array with a multi-million index (which would cause an out of memory error!)

The size of the array can be obtained with the *$#* notation:

```
VSP-8284XSQ:1#% $#list

$#list    = 4

VSP-8284XSQ:1#% @print $list[$#list]
         vars% @print 99
99

VSP-8284XSQ:1#% $list[0]

$list[4]    = 99

VSP-8284XSQ:1#%
```

While the same *$#* and *$'* notations can still be used on individual elements of the array, if they are comma separated values, just as we could do for scalar variables:

```
VSP-8284XSQ:1#% $list[1]

$list[1]    = 1/1-1/10

VSP-8284XSQ:1#% $'list[1]

$list[1]    = 1/1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9,1/10

VSP-8284XSQ:1#% $#list[1]

$#list[1]   = 10

VSP-8284XSQ:1#%
```

Whereas if the array is dereferenced without an index in a CLI command then a comma separated list of the index numbers will be applied.

```
VSP-8284XSQ:1#% @print $list[]
         vars% @print 1,2,3,4
1,2,3,4

VSP-8284XSQ:1#%
```

Note that this property also allows us to iterate through the array in a *@for* loop:

```
@echo off
@for $i &$list[]
    @printf "Index %s has value = %s", $i, $list[$i]
@endfor
@echo on
```

Which produces this output:

```
VSP-8284XSQ:1#%  @echo off
Index 1 has value = 1/1-1/10
Index 2 has value = 1-3
Index 3 has value = newstring
```

```
        Index 4 has value = 99
VSP-8284XSQ:1#%
```

Setting hash variables

In practice hash variables will typically get set by capturing a list of values from the output of some CLI command, and this will be covered in the section below. Nevertheless, a syntax is also available to manually assign a key/value pairs to a hash variable.

```
VSP-8284XSQ:1#% $hash{} = (ports => 2/1-2/5; numbers => 1-10; name => string)

$hash{}     = (name=>string; numbers=>1-10; ports=>2/1-2/5)

VSP-8284XSQ:1#%
```

Each hash element can take the same values and follows the same rules as scalar variables which have been covered in the preceding sections. The order of the hash keys is not necessarily the order in which the hash keys were entered; ACLI will list the hash keys in alphabetical order.

Hash elements can be recalled or overwritten with a new value by specifying the hash key.

```
VSP-8284XSQ:1#% $hash{ports}

$hash{ports} = 2/1-2/5

VSP-8284XSQ:1#% $hash{numbers} = 1-20

$hash{numbers} = 1-20

VSP-8284XSQ:1#% $hash{new} = 99

$hash{new}  = 99

VSP-8284XSQ:1#% $hash{}

$hash{}     = (name=>string; new=>99; numbers=>1-20; ports=>2/1-2/5)

VSP-8284XSQ:1#%
```

The number of key/value pairs in the hash can again be obtained with the *$#* notation:

```
VSP-8284XSQ:1#% $#hash

$#hash      = 4

VSP-8284XSQ:1#%
```

And like with arrays, the same *$#* and *$′* notations can still be used on individual elements of the hash, if they are comma separated values, just as we could do for scalar variables:

```
VSP-8284XSQ:1#% $hash{ports}

$hash{ports} = 2/1-2/5

VSP-8284XSQ:1#% $'hash{ports}

$hash{ports} = 2/1,2/2,2/3,2/4,2/5

VSP-8284XSQ:1#% $#hash{ports}

$#hash{ports} = 5

VSP-8284XSQ:1#%
```

Whereas if the hash is dereferenced without a key in a CLI command then a comma separated list of the keys will be applied.

```
VSP-8284XSQ:1#% @print $hash{}
          vars% @print name,new,numbers,ports
name,new,numbers,ports

VSP-8284XSQ:1#%
```

Note that this property also allows us to iterate through the hash in a *@for* loop:

```
@echo off
@for $key &$hash{}
    @printf "Key '%s' has value = %s", $key, $hash{$key}
@endfor
@echo on
```

Which produces this output:

```
VSP-8284XSQ:1#%  @echo off
Key 'name' has value = string
Key 'new' has value = 99
Key 'numbers' has value = 1-20
Key 'ports' has value = 2/1-2/5
VSP-8284XSQ:1#%
```

Capturing to list & hash variables

This section will look at how we can populate array and hash variables with values directly from the output of a CLI command. All the concepts already covered in the above section where variable capturing was covered for scalar variables still appy here.

As a first example let us capture all our 40G ports and their respective ifIndex into two separate arrays.

```
VSP-8284XSQ:1#% if |40G > $port[],$ifdx[] %1%2
        alias% show interfaces gigabitEthernet interface!!locked |40G > $port[],$ifdx[]
1/41    232    40GbNone         true  false   1950  00:51:00:3f:a8:28 down   down
1/42    236    40GbNone         true  false   1950  00:51:00:3f:a8:2c down   down
2/41    296    40GbNone         true  false   1950  00:51:00:3f:a8:68 down   down
2/42    300    40GbNone         true  false   1950  00:51:00:3f:a8:6c down   down

$port[]     = (1/41; 1/42; 2/41; 2/42)
$ifdx[]     = (232; 236; 296; 300)

VSP-8284XSQ:1#%
```

We could have used scalar variables but the advantage of using arrays is that we can now easily iterate through the values in a *@for* loop. By copy-pasting the following script:

```
@echo off
@for $i &$port[]
    @printf "Port %s has ifIndex %s", $port[$i], $ifdx[$i]
@endfor
@echo on
```

We get the following output:

```
VSP-8284XSQ:1#%  @echo off
Port 1/41 has ifIndex 232
Port 1/42 has ifIndex 236
Port 2/41 has ifIndex 296
Port 2/42 has ifIndex 300
VSP-8284XSQ:1#%
```

This works nicely as we know that both arrays have the exact same number of elements.

An example where arrays come in handy is if we wanted to capture port membership for some different record type, but wanted the port list/ranges not to get merged. For example:

```
PoE-TUNI-Switch:1#% show i-sid > $isids[],$ports[] %1,%4
=================================================================================
                                 Isid Info
=================================================================================
ISID        ISID                    PORT            MLT             ORIGIN
ID          TYPE      VLANID        INTERFACES      INTERFACES
---------------------------------------------------------------------------------
15999003    ELAN_TR   N/A           1/3,1/41        -               CONFIG
15999004    ELAN_TR   N/A           1/4,1/47        -               CONFIG
15999005    ELAN_TR   N/A           1/5,1/46        -               CONFIG
15999006    ELAN_TR   N/A           1/6,1/39        -               CONFIG
15999007    ELAN_TR   N/A           1/7,1/43        -               CONFIG
15999008    ELAN_TR   N/A           1/8,1/42        -               CONFIG
15999009    ELAN_TR   N/A           1/9,1/45        -               CONFIG
15999010    ELAN_TR   N/A           1/10,1/37       -               CONFIG
15999012    ELAN_TR   N/A           1/12,1/38       -               CONFIG
15999014    ELAN_TR   N/A           1/14,1/33       -               CONFIG
15999015    ELAN_TR   N/A           1/15,1/30       -               CONFIG
15999016    ELAN_TR   N/A           1/16,1/29       -               CONFIG
15999017    ELAN_TR   N/A           1/17,1/31       -               CONFIG
15999018    ELAN_TR   N/A           1/18,1/32       -               CONFIG
15999019    ELAN_TR   N/A           1/19,1/44       -               CONFIG
15999025    ELAN_TR   N/A           1/25,1/50       -               CONFIG
15999026    ELAN_TR   N/A           1/26,1/49       -               CONFIG

c: customer vid    u: untagged-traffic
All 17 out of 17 Total Num of i-sids displayed
acli.pl: Displayed Record Count = 17

$isids[]    = (15999003; 15999004; 15999005; 15999006; 15999007; 15999008; 15999009; 1599
$ports[]    = (1/3,1/41; 1/4,1/47; 1/5,1/46; 1/6,1/39; 1/7,1/43; 1/8,1/42; 1/9,1/45; 1/10

PoE-TUNI-Switch:1#%
```

If we had used a scalar variable, the ports would have all got mashed into one combined list.

However a hash is often the most efficient way to capture values and associate them with a key value. The above first example where we captured the all the 40G port numbers and their respective ifIndexes into two separate arrays could have been done using a single hash, as in the following example:

```
VSP-8284XSQ:1#% if |40G > $portIfdx{%1} %2
       alias% show interfaces gigabitEthernet interface!!locked |40G > $portIfdx{%1} %2
1/41    232    40GbNone             true  false   1950  00:51:00:3f:a8:28 down   down
1/42    236    40GbNone             true  false   1950  00:51:00:3f:a8:2c down   down
2/41    296    40GbNone             true  false   1950  00:51:00:3f:a8:68 down   down
2/42    300    40GbNone             true  false   1950  00:51:00:3f:a8:6c down   down

$portIfdx{} = (1/41=>232; 1/42=>236; 2/41=>296; 2/42=>300)

VSP-8284XSQ:1#%
```

Now we have a hash where the key is the port number and the values are the corresponding ifIndexes. Notice how we can specify which column value is to be used as the hash key, by simply using the *%<n>* syntax within the hash curlies. Perhaps we might like to also record the MAC address of each 40G port. This can be easily done by capturing to a second hash, using the same key:

```
VSP-8284XSQ:1#% if |40G > $portIfdx{%1},$portMac{%1} %2%7
       alias% show interfaces gigabitEthernet interface!!locked |40G > $portIfdx{%1},$p
1/41    232    40GbNone             true  false   1950  00:51:00:3f:a8:28 down   down
1/42    236    40GbNone             true  false   1950  00:51:00:3f:a8:2c down   down
2/41    296    40GbNone             true  false   1950  00:51:00:3f:a8:68 down   down
2/42    300    40GbNone             true  false   1950  00:51:00:3f:a8:6c down   down

$portIfdx{} = (1/41=>232; 1/42=>236; 2/41=>296; 2/42=>300)
$portMac{}  = (1/41=>00:51:00:3f:a8:28; 1/42=>00:51:00:3f:a8:2c; 2/41=>00:51:00:3f:a8:68;

VSP-8284XSQ:1#%
```

We could now iterate over our hashes using a similar script, which we will copy-paste to ACLI:

```
@echo off
@for $key &$portIfdx{}
    @printf "Port %s has ifIndex %s and MAC %s", $key, $portIfdx{$key}, $portMac{$key}
@endfor
@echo on
```

Which gives us the following output:

```
VSP-8284XSQ:1#%  @echo off
Port 1/41 has ifIndex 232 and MAC 00:51:00:3f:a8:28
Port 1/42 has ifIndex 236 and MAC 00:51:00:3f:a8:2c
Port 2/41 has ifIndex 296 and MAC 00:51:00:3f:a8:68
Port 2/42 has ifIndex 300 and MAC 00:51:00:3f:a8:6c
VSP-8284XSQ:1#%
```

It therefore becomes possible to create completely new CLI commands, by first extracting the desired information into array or hashes, and then @print-ing it out via a script, all contained in a single command alias!

Another example is if we wanted to have two separate hashes, the 1st one giving us the ifIndex of a given port and the other doing the opposite, i.e. giving us the port number for a given ifIndex:

```
VSP-8284XSQ:1#% if |40G > $portIfdx{%1},$ifdxPort{%2} %2%1
        alias% show interfaces gigabitEthernet interface!!locked |40G > $portIfdx{%1},$i
1/41    232    40GbNone            true  false    1950  00:51:00:3f:a8:28 down    down
1/42    236    40GbNone            true  false    1950  00:51:00:3f:a8:2c down    down
2/41    296    40GbNone            true  false    1950  00:51:00:3f:a8:68 down    down
2/42    300    40GbNone            true  false    1950  00:51:00:3f:a8:6c down    down

$portIfdx{} = (1/41=>232; 1/42=>236; 2/41=>296; 2/42=>300)
$ifdxPort{} = (232=>1/41; 236=>1/42; 296=>2/41; 300=>2/42)

VSP-8284XSQ:1#%
VSP-8284XSQ:1#% $portIfdx{1/42}

$portIfdx{1/42} = 236

VSP-8284XSQ:1#% $ifdxPort{236}

$ifdxPort{236} = 1/42

VSP-8284XSQ:1#%
```

Capturing with regular expressions can be used to capture the hash key on a different line from the actualy hash key value. The ERS *show vlan* output is a perfect example where the VLAN port membership is displayed on the line below the VLAN record. To capture each VLAN's port membership we can do the following:

```
ERS4900-STK#% show vlan > $vlanPorts{%1} '^(\d+)|(\d[\d/,-]+\d)'
Id   Name                 Type     Protocol         PID      Active IVL/SVL Mgmt
---- -------------------- -------- ---------------- ------- ------ ------- ----
1    VLAN #1              Port     None             0x0000  Yes    IVL     No
       Port Members: 1/1-8,1/10,1/12-48,1/50,2/1-8,2/10,2/12-15,2/17-48,3/1-4,3/6-48,3/5
200  VLAN #200            Port     None             0x0000  Yes    IVL     No
       Port Members: 1/9,1/49,2/11,3/5,3/49
201  VLAN #201            Port     None             0x0000  Yes    IVL     No
       Port Members: 1/49,2/11,3/49
209  VLAN #209            Port     None             0x0000  Yes    IVL     Yes
       Port Members: 1/49,2/16,2/50,3/6
210  VLAN #210            Port     None             0x0000  Yes    IVL     No
       Port Members: 1/49,2/9,3/6,3/49
211  VLAN #211            Port     None             0x0000  Yes    IVL     No
       Port Members: 1/49,3/49
220  VLAN #220            Port     None             0x0000  Yes    IVL     No
       Port Members: 1/49,3/5-6,3/49
230  VLAN #230            Port     None             0x0000  Yes    IVL     No
       Port Members: 1/49,3/6,3/49
240  VLAN #240            Port     None             0x0000  Yes    IVL     No
```

```
          Port Members: 1/11,1/49,3/49
Total VLANs: 9

$vlanPorts{} = (1=>1/1-8,1/10,1/12-48,1/50,2/1-8,2/10,2/12-15,2/17-48,3/1-4,3/6-48,3/50;

ERS4900-STK#%
```

Another similar challenging output is the VSP *"show i-sid"* output. Follows an example using regular expressions to capture the ports by I-SID:

```
VSP-8284XSQ:1#% show i-sid > $isidPorts{%1} '(^\d+)\s+\S+\s+\S+\s+[uc\d]+:(\d[\d/,-]+\d)|
================================================================================
                                   Isid Info
================================================================================
ISID         ISID                     PORT            MLT            ORIGIN
ID           TYPE      VLANID         INTERFACES      INTERFACES
--------------------------------------------------------------------------------
666666       ELAN      N/A            c10:1/11,       -              CONFIG
                                      1/13-1/15,1/17,
                                      1/19-1/21,1/23
777777       ELAN      N/A            c20:1/10-1/15,  -              CONFIG
                                      1/25

c: customer vid    u: untagged-traffic
All 2 out of 2 Total Num of i-sids displayed

$isidPorts{} = (666666=>1/11,1/13-1/15,1/17,1/19-1/21,1/23; 777777=>1/10-1/15,1/25)

VSP-8284XSQ:1#%
```

Querying user to set variable

Another way to assign values to variables is to prompt the user for a value. ACLI offers the *'@vars prompt'* embedded command to do this.

```
@vars prompt [optional] [ifunset] <$variable> ["Text to prompt user with"]
```

If *'optional'* is specified, then the user can chose to not provide any value by just hitting enter. In which case, if the variable supplied was already set, it will be unset (deleted). Also, in scripting mode, script execution will continue if user hits enter with no input. If instead the 'optional' argument is not specified then the user is expected to enter a value and if nothing is entered then ACLI will come out of scripting mode.

If *'ifunset'* is specified, then the user is queried for a value only if the variable in question does not exist (i.e. it is not set yet). If the variable is already set, then '@vars prompt' does nothing.

If no *"Text to prompt user with"* is provided, the user will be prompted with a standard message to set the named variable. The '@vars prompt' is typically used in ACLI scripts to obtain user input, as such the user need not know about the variable names used by the script, and it is more user friendly to just ask the user for the information required.

The *<$variable>* will typically be a regular scalar variable. It can also be a hash or array variable, but in this case it has to be for either a specific hash key or a specific array element number, because only a single value can be entered by the user.

A few examples. We will use the following script:

```
@vars prompt $input
$input
```

To test the script, simply copy-paste it into ACLI

```
VSP-8284XSQ:1#% @vars prompt $input
Please enter a value for $input : 42
VSP-8284XSQ:1#% $input

$input       = 42

VSP-8284XSQ:1#%
```

A value was supplied, and the script executed the second and last line, to show the variable.

```
VSP-8284XSQ:1#% @vars prompt $input
Please enter a value for $input :
VSP-8284XSQ:1#%
```

The same script was executed, but this time user simply hit return and provided no value. This is unexpected so ACLI exits from scripting mode. Note that the second line did not execute.

Let's change the script to this:

```
@vars prompt optional $input
$input
```

We execute the script, and provide no value.

```
VSP-8284XSQ:1#% @vars prompt optional $input
Please enter a value for $input [enter to unset]:
VSP-8284XSQ:1#% $input

$input       = <undefined>
```

```
VSP-8284XSQ:1#%
```

Notice that the scripts completes execution in this case.

```
VSP-8284XSQ:1#% @vars prompt optional $input
Please enter a value for $input [enter to skip]:  42
VSP-8284XSQ:1#% $input

$input      = 42

VSP-8284XSQ:1#%
```

No difference if a value is provided.

Note, if we wanted to keep prompting the user until a valid value is entered, then it is sufficient to enclose the '@vars prompt' in a '@until' loop. Here's a new script example.

```
@echo off
@loop
        @vars prompt optional $_ "Are you sure you want to continue (y/n) ?"
@until $_ =~ /[Yy](es)?/ || $_ =~ /[Nn]o?/
@printf "User said : %s", $
@echo on
```

The '@echo off' tells ACLI to stop echoing the switch prompts and the script commands as they are executed, so as to remove clutter during script execution (this is covered in the scripting section). Notice that a custom prompt is provided here.

```
VSP-8284XSQ:1#% @echo off
Are you sure you want to continue (y/n) ?
Are you sure you want to continue (y/n) ?
Are you sure you want to continue (y/n) ?
Are you sure you want to continue (y/n) ?
Are you sure you want to continue (y/n) ? n
User said : n
VSP-8284XSQ:1#%
```

Note that now, the user has no choice but to provide a valid answer, either yes or no (user can still break out of the script using CTRL-C). The answer is held in the default variable '$_' in this example.

Let's change our script back to the simple example, but this time with the *'ifunset'* argumnet:

```
@vars prompt ifunset $input
$input
```

Which gives:

```
VSP-8284XSQ:1#% @vars prompt ifunset $input
VSP-8284XSQ:1#% $input

$input      = 42

VSP-8284XSQ:1#%
```

Notice that we were not even prompted for a value for *$input*, since the variable was already set in our case.

Reserved variables

The ACLI terminal has some reserved variables which cannot be used as user variables. These variable are typically read-only (except exceptions!) and their values are set by ACLI.

- **$$** : Device system name. Same as attribute variable '$_sysname' when available else ACLI automatically extracts the switch name from the device's CLI prompt (this happens only on PassportERS Standby CPUs). Because this variable is often used to redirect output to a file named after the switch (e.g. *$$.cfg*) if the switch name includes any special character which cannot be used in a filename, these characters will be replaced with an underscore '_'. (To get the unmodified switch hostname, consider using the attribute variable '$_sysname' as an alternative).

```
VSP-8284XSQ:1#% $$

$$             = VSP-8284XSQ

VSP-8284XSQ:1#%
```

- **$%** : Switch index number. If ACLI sessions are tied together using socket functionality and the switch prompts (names) are numbered (e.g. Switch-1, Switch-2, Switch-3, etc...) then on each ACLI session '$%' will take value = 1, 2, 3, etc..
- **$@** : Last error message. If last switch CLI command generated an error message, this variable holds the error message. This is useful in ACLI scripting mode, when error detection is disabled ('@error disable') and the script needs to figure out if the previous command failed, without terminating the script. This variable can be primed, i.e. user can set a value to it. Note however that this value is set or reset after every CLI command sent to the connected device, so it needs to be checked immediately after the CLI command we want to check for error messages.

```
VSP-8284XSQ:1#% doctor Foster went to Gloucester
                     ^
% Invalid input detected at '^' marker.
VSP-8284XSQ:1#% $@

$@             = Invalid input detected at ^ marker.

VSP-8284XSQ:1#% pwd
/intflash
VSP-8284XSQ:1#% $@

$@             = <undefined>

VSP-8284XSQ:1#%
```

- **$>** : Last CLI prompt from connected device. Useful to create logic to move between different CLI exec levels using ACLI scripts.
- **$<number>** : When sourcing a script with *'@source'* or *'@run'*, holds positional argument number provided to script.
- **$*** : When sourcing a script with *'@source'* or *'@run'*, holds concatenated arguments provided to script.
- **$ALL** : This variable will always translate to all the ethernet ports on the connected device.
- **$<number>/ALL** : This variable will always translate to all the ethernet ports on the specified slot number of the connected device. For example $1/ALL will translate to all ports on slot 1.
- **$<number>:ALL** : Same as above, but using the ExtremeXOS slot:port notation, will translate to all the ethernet ports on the specified slot number of the connected device. For example $2:ALL will translate to all ports on slot 2.

Attribute variables

ACLI's attribute variables are read-only variables which make available in ACLI the underlying Control::CLI::Extreme Perl module attributes. http://search.cpan.org/~lstevens/Control-CLI-Extreme-1.00/lib/Control/CLI/Extreme.pm#Main_I/O_Object_Methods see: *'attribute() - Return device attribute value'*

Attribute variables always commence with '$_' followed by the attribute name. All the available attribute variables can be easily dumped with the embedded *'@vars atribute'* command.

```
VSP-8284XSQ:1#% @vars attribute

$_family_type         = PassportERS
$_is_nncli            = 1
$_is_acli             = 1
$_model               = VSP-8284-XSQ
$_sw_version          = 8.0.0.0
$_fw_version          = 8.0.0.0
$_slots               = 1,2
$_ports               = ,ARRAY(0x411e43c),ARRAY(0x41238e4)
$_sysname             = VSP-8284XSQ
$_base_mac            = 00-51-00-3f-a8-00
$_baudrate            =
$_max_baud            = 115200
$_is_voss             = 1
$_is_apls             = 0
$_apls_box_type       =
$_brand_name          = Extreme Networks
$_is_master_cpu       = 1
$_is_dual_cpu         = 0
$_cpu_slot            = 1
$_is_ha               =
$_stp_mode            = mstp
$_oob_ip              = 192.168.56.80
$_oob_virt_ip         =
$_oob_standby_ip      =
$_is_oob_connected    = 1

VSP-8284XSQ:1#%
```

The above command might require a small delay to complete. This has to do with how Control::CLI::Extreme handles the attributes; some are discovered & set during connection, others are set on demand, the first time a query is made for the attribute (in this case the modules executes switch CLI commands against the switch to obtain the data; these commands will not be visible on the ACLI interface). Though once an attribute has been fetched, it is cached and quickly retrievable on subsequent requests.

To view the *$_ports* arrays, simply provide the relevant slot number (from *$_slots*) in [] brackets:

```
VSP-8284XSQ:1(config)#% $_ports[1]

$_ports[1]  = 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,

VSP-8284XSQ:1(config)#% $_ports[2]

$_ports[2]  = 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,

VSP-8284XSQ:1(config)#%
```

Viewing variables

Variables of any type, user defined, reserved and attribute variables can be viewed by simply typing the variable on the ACLI session.

```
VSP-8284XSQ:1#% $myvar

$myvar      = 1/1,1/5

VSP-8284XSQ:1#% $$

$$          = VSP-8284XSQ

VSP-8284XSQ:1#% $_sysname

$_sysname   = VSP-8284XSQ

VSP-8284XSQ:1#%
```

User defined variables can also be dumped via the embedded *'@vars show'* or *'@vars raw'* commands. The former can also be abbreviated as just *'@vars'* or even just *'@$'* and the latter as *'@$ raw'*.

```
VSP-8284XSQ:1#% @vars show

$_            = 10
$allPorts     = 1/1-1/42,2/1-2/42
$hash{}       = (name=>string; new=>99; numbers=>1-20; ports=>2/1-2/5)
$i            = 4
$ifdx[]       = (232; 236; 296; 300)
$ifdxPort{}   = (232=>1/41; 236=>1/42; 296=>2/41; 300=>2/42)
$input        = 42
$isidPorts{}  = (666666=>1/11,1/13-1/15,1/17,1/19-1/21,1/23; 777777=>1/10-1/15,1/25)
$key          = 2/42
$list[]       = (1/1-1/10; 1-3; newstring; 99)
$myvar        = 1/1,1/5
$port[]       = (1/41; 1/42; 2/41; 2/42)
$portIfdx{}   = (1/41=>232; 1/42=>236; 2/41=>296; 2/42=>300)
$portMac{}    = (1/41=>00:51:00:3f:a8:28; 1/42=>00:51:00:3f:a8:2c; 2/41=>00:51:00:3f:a8:6
$portRange    = 1/1-1/10
$range        = 1-10

Unsaved variables exist

VSP-8284XSQ:1#% @$

$_            = 10
$allPorts     = 1/1-1/42,2/1-2/42
$hash{}       = (name=>string; new=>99; numbers=>1-20; ports=>2/1-2/5)
$i            = 4
$ifdx[]       = (232; 236; 296; 300)
$ifdxPort{}   = (232=>1/41; 236=>1/42; 296=>2/41; 300=>2/42)
$input        = 42
$isidPorts{}  = (666666=>1/11,1/13-1/15,1/17,1/19-1/21,1/23; 777777=>1/10-1/15,1/25)
$key          = 2/42
$list[]       = (1/1-1/10; 1-3; newstring; 99)
$myvar        = 1/1,1/5
$port[]       = (1/41; 1/42; 2/41; 2/42)
$portIfdx{}   = (1/41=>232; 1/42=>236; 2/41=>296; 2/42=>300)
$portMac{}    = (1/41=>00:51:00:3f:a8:28; 1/42=>00:51:00:3f:a8:2c; 2/41=>00:51:00:3f:a8:6
$portRange    = 1/1-1/10
$range        = 1-10

Unsaved variables exist

VSP-8284XSQ:1#%
```

If some variables have unsaved values, a message is displayed indicating so. See *"Saving device related ACLI settings"* section covering the *'@save'* embedded command.

The same *'@vars'* command also accepts wildcards to only display a subset or individual variable.

```
VSP-8284XSQ:1#% @vars show $input

$input      = 42

VSP-8284XSQ:1#% @vars show ange

$portRange  = 1/1-1/10
$range      = 1-10

VSP-8284XSQ:1#%
```

Whereas to dump all attribute variables use the *'@vars attribute'* command, which also accepts wildcards.

```
VSP-8284XSQ:1#% @vars attribute ip

$_oob_ip            = 192.168.56.71
$_oob_virt_ip       =
$_oob_standby_ip    =

VSP-8284XSQ:1#%
```

Dereferencing variables in CLI commands

Once a variable has been set it can be used in any ACLI command (CLI commands, alias arguments, etc..) and they technically can even be used as commands themselves (instead of aliases) but this is not their prime purpose.

Variables will be dereferenced if enclosed in double quotes or curly brackets '{}'. Variables are not dereferenced if enclosed in single quotes.

To dereference a variable in a CLI command simply place the variable where it is needed.

```
VSP-8284XSQ:1#% show interfaces gigabitEthernet interface $myvar
        vars% show interfaces gigabitEthernet interface 1/1,1/5
================================================================================
                                Port Interface
================================================================================
PORT                         LINK  PORT          PHYSICAL        STATUS
NUM      INDEX DESCRIPTION   TRAP  LOCK    MTU   ADDRESS         ADMIN  OPERATE
--------------------------------------------------------------------------------
1/1      192   10GbNone      true  false   1950  00:51:00:ca:e0:00 down   down
1/5      196   10GbNone      true  false   1950  00:51:00:ca:e0:04 down   down

VSP-8284XSQ:1#% if $myvar
         vars% if 1/1,1/5
        alias% show interfaces gigabitEthernet interface 1/1,1/5!!locked
================================================================================
                                Port Interface
================================================================================
PORT                         LINK  PORT          PHYSICAL        STATUS
NUM      INDEX DESCRIPTION   TRAP  LOCK    MTU   ADDRESS         ADMIN  OPERATE
--------------------------------------------------------------------------------
1/1      192   10GbNone      true  false   1950  00:51:00:ca:e0:00 down   down
1/5      196   10GbNone      true  false   1950  00:51:00:ca:e0:04 down   down
VSP-8284XSQ:1#%
```

If variable echoing is enabled, a line is added after the comand entered showing the variable substitution. Variable echoing is by default enabled but can be disabled using the *'@vars echo'* embedded command or the *'vars echo'* command under ACLI control interface.

If the variable is separated by spaces or non alphanumeric characters, the above will work fine. If instead the variable needs to be dereferenced right up against some other alphanumeric text then the variable will need to be enclosed in curly brackets '{}', as in the example below.

```
; $byte = Byte value to use in Nick-name + BMAC

router isis
   manual-area 49.0000
   spbm 1
   spbm 1 b-vid 4051-4052 primary 4051
   spbm 1 nick-name 0.00.$byte
   system-id 00bb.0000.{$byte}00
exit
```

In the example above, if $byte was not enclosed in curlies, ACLI would try and look for a variable named $byte00, which would not exist. Use of curlies '{}' is actually embedding Perl code. See *"Eval of Perl in CLI commands"* section.

There is no difference between dereferincing an existing variable in user interactive use and in ACLI scripting mode. However, if trying to dereference a variable which is not set (i.e. does not exist) then the behaviour is different depending on whether ACLI is running in scripting mode or not.

In user interactive mode, an error is shown because the variable is not set; the variable is thus not replaced and the command is sent as is to the switch, which will also complain:

```
VSP-8284XSQ:1#% show mlt $nosuchvar
          vars% <variable $nosuchvar is undefined>
          vars% show mlt $nosuchvar
                        ^
% Invalid input detected at '^' marker.
VSP-8284XSQ:1#%
```

To demonstrate scripting mode, we shall copy-paste the following simple script:

```
@if $nosuchvar
    show mlt $nosuchvar
@else
    show mlt $nosuchvar
@endif
```

Which gives the following:

```
VSP-8284XSQ:1#%  @if $nosuchvar
          vars% <variable $nosuchvar is undefined>
VSP-8284XSQ:1#%  @else
VSP-8284XSQ:1#%      show mlt $nosuchvar
          vars% <variable $nosuchvar is undefined>
          vars%     show mlt ''
                            ^
% Invalid input detected at '^' marker.
VSP-8284XSQ:1#%
```

We still get an error indicating that the variable is not set. However in scripting mode undefined variables will always be dereferenced as the empty string ''. This is because, if used in ACLI's scripting conditional operators (@if, @while, @until, etc..) the undefined variable will need to be handed off to Perl for evaluation and it needs to be seen by Perl as an empty string (else the Perl eval would fail). Also in scripting mode, use of an undefined variable will result in script execution to stop, as seen above (the last @endif statement never executes).

Using variables in scripts

As we have seen, all the variables types can be used in CLI commands, either by directly embedding them in CLI commands or via the embedded Perl code snippets using curly brackets '{}'.

In ACLI scripting mode, all variables can also be used in ACLI's conditional operators, which are covered in the scripting section. A quick list of these operators follows.

```
@if <cond>, @elsif <cond>, @else, @endif        if / elsif / else conditional operato
@while <cond>, @endloop                         while loop construct
@loop, @until <cond>                            loop until construct
@for <$var> &<start>..<end>[:<step>], @endfor   for loop construct using range input
@for <$var> &<comma-separated-list>, @endfor    for loop construct using list input
@next [if <cond>]                               jump to next value in a for loop cons
@last [if <cond>]                               break out of a while, until or for lo
@exit [if <cond>]                               break out of sourced script
```

Variables can be used wherever the *'<cond>'* field is seen as well as after the '&' character in the *'@for'* loop operator. The conditions been matched with the variables do not need to be enclosed in any curly brackets here. The *'<cond>'* condition matches are also eval-ed as Perl code, so parenthesis can of course be used if needed depending on use of logical operators. See the scripting section.

When running scripts it can be necessary to use a number of different variables to make the script function. Once the script has terminated these variables will remain visible in the *@vars show* commands and will in some way pollute any variables which may have been set for interactive use of the CLI. To prevent script variables from polluting the other non-script variables there are two possibilities.

Either insert the following embedded command at the very end of the script:

```
@vars clear script
```

Any variable which was brought into existence during execution of a script will have an internal marker; the above command will thus delete all these variables. The problem with this approach is that the script could terminate before reaching the end, the above command does not get executed and the script variables remain. Of course the above command can be executed by the user.

The other, and more recent, approach is to declare the variables withing the script using the *@my* command. There are three possible forms:

```
@my $var = 1/1
@my $var1, $var2, $h{}, $l[]
@my $prfx_*
```

The first form declares one variable and at the same time initializes that variable with a value. The second form can declare a comma separated list of variables, but these cannot be initialized to a value in this form. The third and last form declares that any variable starting with *$prfx_* will be treated as internal to the script. Hence if we were writing a script called init.run, we could include a declaration of:

```
@my $init_*
```

And then within the script if we spawn variables such as *$init_var1, $init_var2, $init_list[], $init_hash{}* these will automatically be considered script variables and will not be visisble in the *@vars show* command.

Dictionary variables

The ACLI dictionary functionality also makes use of variables and these are marked so as to be kept separate from user variables. These can be viewed with *@vars show dictionary*. See the *"Dictionaries"* section for more information.

Storing variables

User defined variables can be saved against the MAC address of the device to which the ACLI session is connected to. This is useful for automatically restoring all the same variables when re-connecting via ACLI to the same device again. To save the variables use either the *'@save all'* or *'@save vars'* embedded commands. See also *"Saving device related ACLI settings"* section.

Repeating a Command

There are two ways to repeat a command with the ACLI terminal, in interactive mode.

A command can be repeated indefinietly by simply appending to it the *'@'* character followed by an optional number representing the delay seconds to wait between every command execution. If no optional number of seconds is provided, then the commands are repeated as fast as possible (i.e. as soon as a new CLI prompt is received from the connected host). To break out of the loop simply hit any key on the terminal.

- \<CLI-command-to-repeat> **@ \<interval-seconds>**

```
VSP-8284XSQ:1#% show clock @5
Mon Jul 30 13:36:43 2018 UTC


Mon Jul 30 13:36:47 2018 UTC


Mon Jul 30 13:36:52 2018 UTC
```

This will also work with multiple commands separated by semicolons. All the commands will be repeated.

A command can also be repeated for a precise number of times by appending to it the *'&'* character followed by the desired iteration sequence. The value of the iteration sequence can also be embedded in the command being executed, using the same formatting as Perls' *sprintf()*. The sequence syntax can take a number of formats:

- \<CLI-command-to-repeat-with-optional-embedded-%s> **&\<start&>..&\<end>[:&\<step>] ...**
- \<CLI-command-to-repeat-with-optional-embedded-%s> **&[']\<comma-separated-list-or-port-range> ...**

This will also work with multiple commands separated by semicolons. All the commands will be repeated.

A few examples will help illustrate; the first exaple creates 9 mlt instances, numbered 1..9:

```
VSP-8284XSQ:1(config)#% mlt %s &1..9
VSP-8284XSQ:1(config)#% mlt 1
VSP-8284XSQ:1(config)#% mlt 2
VSP-8284XSQ:1(config)#% mlt 3
VSP-8284XSQ:1(config)#% mlt 4
VSP-8284XSQ:1(config)#% mlt 5
VSP-8284XSQ:1(config)#% mlt 6
VSP-8284XSQ:1(config)#% mlt 7
VSP-8284XSQ:1(config)#% mlt 8
VSP-8284XSQ:1(config)#% mlt 9
VSP-8284XSQ:1(config)#%
```

The next example also creates 9 mlts, but their ids are now 10,20,30...90:

```
VSP-8284XSQ:1(config)#% mlt %s &10..90:10
VSP-8284XSQ:1(config)#% mlt 10
VSP-8284XSQ:1(config)#% mlt 20
VSP-8284XSQ:1(config)#% mlt 30
VSP-8284XSQ:1(config)#% mlt 40
VSP-8284XSQ:1(config)#% mlt 50
VSP-8284XSQ:1(config)#% mlt 60
VSP-8284XSQ:1(config)#% mlt 70
VSP-8284XSQ:1(config)#% mlt 80
VSP-8284XSQ:1(config)#% mlt 90
VSP-8284XSQ:1(config)#%
```

If we now wanted to delete all these mlts, we could simply use a list:

```
VSP-8284XSQ:1(config)#% no mlt %s &'1-10,20,30,40,50,60,70,80,90
VSP-8284XSQ:1(config)#% no mlt 1
VSP-8284XSQ:1(config)#% no mlt 2
VSP-8284XSQ:1(config)#% no mlt 3
VSP-8284XSQ:1(config)#% no mlt 4
VSP-8284XSQ:1(config)#% no mlt 5
VSP-8284XSQ:1(config)#% no mlt 6
VSP-8284XSQ:1(config)#% no mlt 7
VSP-8284XSQ:1(config)#% no mlt 8
VSP-8284XSQ:1(config)#% no mlt 9
VSP-8284XSQ:1(config)#% no mlt 10
VSP-8284XSQ:1(config)#% no mlt 20
VSP-8284XSQ:1(config)#% no mlt 30
VSP-8284XSQ:1(config)#% no mlt 40
VSP-8284XSQ:1(config)#% no mlt 50
VSP-8284XSQ:1(config)#% no mlt 60
VSP-8284XSQ:1(config)#% no mlt 70
VSP-8284XSQ:1(config)#% no mlt 80
VSP-8284XSQ:1(config)#% no mlt 90
VSP-8284XSQ:1(config)#%
```

Note that since our list includes a range, we must add the ' character to expand the 1-10 range into a list as well

Or if we had captured that list in a variable $mltids:

```
VSP-8284XSQ:1(config)#% $mltids

$mltids      = 1-10,20,30,40,50,60,70,80,90

VSP-8284XSQ:1(config)#% no mlt %s &'$mltids
              vars% no mlt %s &'1-10,20,30,40,50,60,70,80,90
VSP-8284XSQ:1(config)#% no mlt 1
VSP-8284XSQ:1(config)#% no mlt 2
VSP-8284XSQ:1(config)#% no mlt 3
VSP-8284XSQ:1(config)#% no mlt 4
VSP-8284XSQ:1(config)#% no mlt 5
VSP-8284XSQ:1(config)#% no mlt 6
VSP-8284XSQ:1(config)#% no mlt 7
VSP-8284XSQ:1(config)#% no mlt 8
VSP-8284XSQ:1(config)#% no mlt 9
VSP-8284XSQ:1(config)#% no mlt 10
VSP-8284XSQ:1(config)#% no mlt 20
VSP-8284XSQ:1(config)#% no mlt 30
VSP-8284XSQ:1(config)#% no mlt 40
VSP-8284XSQ:1(config)#% no mlt 50
VSP-8284XSQ:1(config)#% no mlt 60
VSP-8284XSQ:1(config)#% no mlt 70
VSP-8284XSQ:1(config)#% no mlt 80
VSP-8284XSQ:1(config)#% no mlt 90
VSP-8284XSQ:1(config)#%
```

Again, when dereferencing a $variable, which might produce ranges, we need to either place the ' character after the '&' operator, or alternatively the variable can be dereferenced in raw list mode with $'variable:

```
VSP-8284XSQ:1(config)#% no mlt %s &$'mltids
              vars% no mlt %s &1,2,3,4,5,6,7,8,9,10,20,30,40,50,60,70,80,90
VSP-8284XSQ:1(config)#% no mlt 1
VSP-8284XSQ:1(config)#% no mlt 2
VSP-8284XSQ:1(config)#% no mlt 3
VSP-8284XSQ:1(config)#% no mlt 4
VSP-8284XSQ:1(config)#% no mlt 5
VSP-8284XSQ:1(config)#% no mlt 6
VSP-8284XSQ:1(config)#% no mlt 7
VSP-8284XSQ:1(config)#% no mlt 8
VSP-8284XSQ:1(config)#% no mlt 9
VSP-8284XSQ:1(config)#% no mlt 10
VSP-8284XSQ:1(config)#% no mlt 20
VSP-8284XSQ:1(config)#% no mlt 30
VSP-8284XSQ:1(config)#% no mlt 40
VSP-8284XSQ:1(config)#% no mlt 50
```

```
VSP-8284XSQ:1(config)#% no mlt 60
VSP-8284XSQ:1(config)#% no mlt 70
VSP-8284XSQ:1(config)#% no mlt 80
VSP-8284XSQ:1(config)#% no mlt 90
VSP-8284XSQ:1(config)#%
```

Multiple ranges can also be specified, of either type, separated by space; however in this case the ranges need to be consistent (i.e. have the same number of iterations) otherwise the command will not be accepted

```
VSP-8284XSQ:1(config)#% mlt %s member %s &1..9 1/1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9
VSP-8284XSQ:1(config)#% mlt 1 member 1/1
VSP-8284XSQ:1(config)#% mlt 2 member 1/2
VSP-8284XSQ:1(config)#% mlt 3 member 1/3
VSP-8284XSQ:1(config)#% mlt 4 member 1/4
VSP-8284XSQ:1(config)#% mlt 5 member 1/5
VSP-8284XSQ:1(config)#% mlt 6 member 1/6
VSP-8284XSQ:1(config)#% mlt 7 member 1/7
VSP-8284XSQ:1(config)#% mlt 8 member 1/8
VSP-8284XSQ:1(config)#% mlt 9 member 1/9
VSP-8284XSQ:1(config)#%
```

In all the above examples, the values are dereferenced as a simple text string (*%s*) in the command string. However any of Perl's *sprintf()* format conversions are accepted: https://perldoc.perl.org/functions/sprintf.html

The repeat command functions will not only work on single CLI commands but can also work on multiple commands, if these are concatenated with semi-colon ';'. In the above example, mlt port membership is set across 9 separate mlts; however it assumed that the mlts were already created. If one wanted to create the mlts and assign the port memebership at the same time, one could do the following:

```
VSP-8284XSQ:1(config)#% mlt %s; mlt %s member %s; &1..9 1..9 1/1,1/2,1/3,1/4,1/5,1/6,1/7,
VSP-8284XSQ:1(config)#% mlt 1; mlt 1 member 1/1
VSP-8284XSQ:1(config)#% mlt 1
VSP-8284XSQ:1(config)#% mlt 1 member 1/1
VSP-8284XSQ:1(config)#% mlt 2; mlt 2 member 1/2
VSP-8284XSQ:1(config)#% mlt 2
VSP-8284XSQ:1(config)#% mlt 2 member 1/2
VSP-8284XSQ:1(config)#% mlt 3; mlt 3 member 1/3
VSP-8284XSQ:1(config)#% mlt 3
VSP-8284XSQ:1(config)#% mlt 3 member 1/3
VSP-8284XSQ:1(config)#% mlt 4; mlt 4 member 1/4
VSP-8284XSQ:1(config)#% mlt 4
VSP-8284XSQ:1(config)#% mlt 4 member 1/4
VSP-8284XSQ:1(config)#% mlt 5; mlt 5 member 1/5
VSP-8284XSQ:1(config)#% mlt 5
VSP-8284XSQ:1(config)#% mlt 5 member 1/5
VSP-8284XSQ:1(config)#% mlt 6; mlt 6 member 1/6
VSP-8284XSQ:1(config)#% mlt 6
VSP-8284XSQ:1(config)#% mlt 6 member 1/6
VSP-8284XSQ:1(config)#% mlt 7; mlt 7 member 1/7
VSP-8284XSQ:1(config)#% mlt 7
VSP-8284XSQ:1(config)#% mlt 7 member 1/7
VSP-8284XSQ:1(config)#% mlt 8; mlt 8 member 1/8
VSP-8284XSQ:1(config)#% mlt 8
VSP-8284XSQ:1(config)#% mlt 8 member 1/8
VSP-8284XSQ:1(config)#% mlt 9; mlt 9 member 1/9
VSP-8284XSQ:1(config)#% mlt 9
VSP-8284XSQ:1(config)#% mlt 9 member 1/9
VSP-8284XSQ:1(config)#%
VSP-8284XSQ:1(config)#% show mlt
================================================================================
                                  Mlt Info
================================================================================
              PORT    MLT    MLT       PORT          VLAN
MLTID IFINDEX NAME    TYPE   ADMIN CURRENT  MEMBERS  IDS
--------------------------------------------------------------------------------
1    6144   MLT-1    access  norm   norm     1/1
2    6145   MLT-2    access  norm   norm     1/2
3    6146   MLT-3    access  norm   norm     1/3
4    6147   MLT-4    access  norm   norm     1/4
```

```
5    6148   MLT-5          access   norm   norm     1/5
6    6149   MLT-6          access   norm   norm     1/6
7    6150   MLT-7          access   norm   norm     1/7
8    6151   MLT-8          access   norm   norm     1/8
9    6152   MLT-9          access   norm   norm     1/9


All 9 out of 9 Total Num of mlt displayed
```

The fact that the repeat operator does not expand ranges (without the ' character) comes in useful with with the SLX product family where CLI commands using port ranges are only allowed as long as the ranges do not span slots. The SLX-9850 can have multiple slots. Given a port selection spanning multiple slots, unlike on other Extreme switches, it would not be possible to perform a configuration across them all at once:

```
PE1-9850(config)#% $ports

$ports         = 1/1-6,1/8-13,1/15-20,1/41-46,1/48-53,1/55-60,4/1-72

PE1-9850(config)#% ife $ports
          vars% ife 1/1-6,1/8-13,1/15-20,1/41-46,1/48-53,1/55-60,4/1-72
         alias% interface Ethernet 1/1-6,1/8-13,1/15-20,1/41-46,1/48-53,1/55-60,4/1-72
----------------------------------^
syntax error: "1/1-6,1/8-13,1/15-20,1/41-46,1/48-53,1/55-60,4/1-72" has a bad length/size
PE1-9850(config)#%
```

Whereas using the repeat operator (without the ' character) yields the most efficient command sequence to configure all the ports:

```
PE1-9850(config)#% ife %s; no shut; exit; &$ports
           vars% ife %s; no shut; exit; &1/1-6,1/8-13,1/15-20,1/41-46,1/48-53,1/55-60,4
PE1-9850(config)#% ife 1/1-6; no shut; exit
PE1-9850(config)#% ife 1/1-6
         alias% interface Ethernet 1/1-6
PE1-9850(conf-if-eth-1/1-6)#% no shut
PE1-9850(conf-if-eth-1/1-6)#% exit
PE1-9850(config)#% ife 1/8-13; no shut; exit
PE1-9850(config)#% ife 1/8-13
         alias% interface Ethernet 1/8-13
PE1-9850(conf-if-eth-1/8-13)#% no shut
PE1-9850(conf-if-eth-1/8-13)#% exit
PE1-9850(config)#% ife 1/15-20; no shut; exit
PE1-9850(config)#% ife 1/15-20
         alias% interface Ethernet 1/15-20
PE1-9850(conf-if-eth-1/15-20)#% no shut
PE1-9850(conf-if-eth-1/15-20)#% exit
PE1-9850(config)#% ife 1/41-46; no shut; exit
PE1-9850(config)#% ife 1/41-46
         alias% interface Ethernet 1/41-46
PE1-9850(conf-if-eth-1/41-46)#% no shut
PE1-9850(conf-if-eth-1/41-46)#% exit
PE1-9850(config)#% ife 1/48-53; no shut; exit
PE1-9850(config)#% ife 1/48-53
         alias% interface Ethernet 1/48-53
PE1-9850(conf-if-eth-1/48-53)#% no shut
PE1-9850(conf-if-eth-1/48-53)#% exit
PE1-9850(config)#% ife 1/55-60; no shut; exit
PE1-9850(config)#% ife 1/55-60
         alias% interface Ethernet 1/55-60
PE1-9850(conf-if-eth-1/55-60)#% no shut
PE1-9850(conf-if-eth-1/55-60)#% exit
PE1-9850(config)#% ife 4/1-72; no shut; exit
PE1-9850(config)#% ife 4/1-72
         alias% interface Ethernet 4/1-72
PE1-9850(conf-if-eth-4/1-72)#% no shut
PE1-9850(conf-if-eth-4/1-72)#% exit
PE1-9850(config)#%
```

Note that using &' would have also worked, but would have resulted in "no shut" being executed individually on every single port after expanding the port ranges, which would have been less efficient. Also note that this example

only works if the ACLI terminal portrange spanslot mode is disabled; see ACLI Control "terminal portrange spanslots" and/or the 'default_port_range_mode' key in *acli.ini*

CLI augmented switches

The ACLI terminal, in interactive mode, allows some standard switches to be added to regular CLI commands of the connected device. These are listed below:

- **-y** : On CLI commands where the connected host will ask for a confirmation prompt (e.g. "Are you sure (Y/N) ?"), ACLI will automatically feed a 'Y' to the confirmation prompt. Note that in ACLI scripting mode this is done automatically and there is no need to add a *'-y'* switch to the commands. The exception is if the confirmation prompt contains either the 'reset' or 'reboot' keywords; in this case scripting mode will not automatically feed a 'Y', unless a *'-y'* was added to the command, or to the alias which triggered the script sequence.
- **-n** : On CLI commands where the connected host will ask for a confirmation prompt (e.g. "Are you sure (Y/N) ?"), ACLI will automatically feed a 'N' to the confirmation prompt. Perhaps not very useful, but available for completeness!
- **-e** : On CLI commands with ACLI redirect to file, adding the *'-e'* switch will ensure that the command output is also echo-ed to the terminal session (not just redirected to file).
- **-s** : On CLI commands followed by an ACLI grep pattern, determines whether the grep patterns should be treated as case-sensitive.
- **-i[n]** : Perform indentation and unwrapping of command output. In practice this switch is only applicable to certain switch commands:
  - **show running-config** : on PassportERS/VOSS and BaystackERS devices which have a configuration file format which uses configuration contexts but which (unfortunately) do not provide any indentation. Applying the *'-i'* switch will thus add indentation to the configuration file. By default 3 space characters are used for indentation (or whatever has been set in *acli.ini*). Alternatively a number n can be provided with the *'-i[n]'* switch and then the indentation will be done using n space characters. In addition to that, on BaystackERS, the *'-i'* switch will also perform

unwrapping of any config line (longer than 131 characters) which the ERS switch would otherwise wrap.
- ○ **show log file** & **show logging** : On BaystackERS and ISW devices the *'-i'* switch will perform unwrapping of long lines which the connected device has broken into two or more lines (which is not acceptable if we need to grep these lines with ACLI).
- **-b** : Remove comment/banner lines from show configuration output. This switch is essentially performing a simple negative grep on lines beginning with the character designating a comment line (';' on BaystackERS and '#' on most other devices). In practice this switch is only useful with switch commands which display the config file as it allows to obtain a compact config file which contains no comment/banner lines.
- **-o[n]** : While socket tied and sourcing (ACLI scripting mode), send command to socket with optional [n] delay in seconds
- **-f** : On commands feeding input data (<CLI command> -f // <input1> // <input2> ...) allows the input data to be cached for future invocation of the same command on same family type device. The cached input data is stored in file *acli.cache*
- **-h** : On commands feeding input data (<CLI command> -h // <input1> // <input2> ...) allows the input data to be cached for future invocation of the same command on same device. The cached input data is stored in file *acli.cache*
- **-peercpu** : On PassportERS/VOSS chassis based devices with dual CPs allows a CLI command entered on a session to the Master CP to be actually executed on the Standby CP alone (see the Peer CP functionality). Switch can also be abbreviated to *-peer*
- **-bothcpus** : On PassportERS/VOSS chassis based devices with dual CPs allows a CLI command entered on a session to the Master CP to be executed on both the Master CP and the Standby CP simultaneously (in practice the commands is executed on the Standby CP slightly before it is executed on the Master CP, so as to work to reset both CPs in a HA-mode config)(see the Peer CP functionality). Switch can also be abbreviated to *-both*

Note that the *'-i'* & *'-b'* switches are usually used together and are always appended to the "show running-config" CLI command which renders the config output in a format which is suitable for performing grep on if needed. To make things easy, the pre-defined ACLI alias *'cfg'* does just that:

```
VSP-8284XSQ:1#% cfg
        alias% show running-config -ib
config terminal
boot config flags sshd
boot config flags telnetd
password password-history 3
ssh
no web-server secure-only
interface mgmtEthernet mgmt
   auto-negotiate
   ip address 192.168.56.71 255.255.255.0
exit
qos queue-profile 1 member add 1/1-1/42,2/1-2/42
no ntp
end

VSP-8284XSQ:1#%
```

Eval of Perl in CLI commands

With ACLI in interactive mode it is possible to eval Perl code snippets embedded in CLI command lines by enclosing them in curly brackets '{}'. The use case, to date, is for dereferencing $variables using Perl's sprintf() function. Follows an example ACLI script used by the author to set up SPB on a VSP node in a lab environment.

```
; $node = Node ID must be set
; $nni  = List of NNI ports
;
spbm
router isis
   manual-area 49.0000
   spbm 1
   spbm 1 b-vid 4051-4052 primary 4051
   spbm 1 nick-name 0.00.{sprintf "%02d", $node}
   system-id 00bb.0000.{sprintf "%02d", $node}00
exit
vlan create 4051 name "B-VLAN-1" type spbm-bvlan
vlan create 4052 name "B-VLAN-2" type spbm-bvlan

vlan members remove 1 $nni
interface gigabitEthernet $nni
   isis
   isis spbm 1
   isis enable
exit

cfm spbm mepid $node
cfm spbm enable
router isis enable
```

The variable $node is given a decimal value (1-99). When using this variable to derive a nick-name and the system-id it is desired for the variable to be encoded as a 2 digit decimal number. This means that if $node = 1-9 we have to use 01-09 in the nick-name and system-id fields. A hex number could also easily be produced using sprintf() with "%02x".

Another example is this other ACLI script to enable vIST on an already configured SPB node (Note, this works on the assumption that one of the vIST peers has an even $node and the other vIST peer has an odd $node number. The same script is simply executed on both nodes).

```
; $node   = Node ID must be set
;
no router isis enable
router isis
   spbm 1 smlt-peer-system-id 00bb.0000.{sprintf "%02d", $node & 1 ? $node + 1 : $node -
   spbm 1 smlt-virtual-bmac 00:bb:00:00:{sprintf "%02d", $node & 1 ? $node : $node - 1}:f
exit
vlan create 4000 name "IST-VLAN" type port-mstprstp 0
vlan i-sid 4000 1000{sprintf "%02d", $node & 1 ? $node : $node - 1}{sprintf "%02d", $node
interface Vlan 4000
   ip address 192.168.255.{sprintf "%02d", $node & 1 ? 1 : 2} 255.255.255.252
exit
virtual-ist peer-ip 192.168.255.{sprintf "%02d", $node & 1 ? 2 : 1} vlan 4000
router isis enable
```

In this example Perl's conditional assignment is combined with sprintf().

Another reason for using curlies is to dereference a variable which otherwise would be delimited by alphanumeric characters.

```
; $byte = Byte value to use in Nick-name + BMAC

router isis
```

```
        manual-area 49.0000
        spbm 1
        spbm 1 b-vid 4051-4052 primary 4051
        spbm 1 nick-name 0.00.$byte
        system-id 00bb.0000.{$byte}00
exit
```

In the example above, if $byte was not enclosed in curlies, ACLI would try and look for a variable named $byte00, which would not exist. There is no visible Perl code here, though what happens is that the value of $byte is eval-ed as Perl code, which, surprise, returns the same value!

Another ACLI script example heavily using curlies:

```
# Create 500 IPv4 interfaces (with VRRP) + 500 IPv6 interfaces (with VRRP)
ipv6 forwarding
@for $blk &1..2
        @for $vln &0..255
                vlan create {sprintf "%d%03d", $blk, $vln} type port-mstprstp 0
                interface vlan {sprintf "%d%03d", $blk, $vln}
                    ip address 10.$blk.$vln.1/24
                    ip vrrp version 3
                    ip vrrp address 1 10.$blk.$vln.254
                    ip vrrp 1 enable
                    ipv6 interface enable
                    ipv6 interface address 300$blk:{sprintf "%02x", $vln}::$node/64
                    ipv6 forwarding
                    ipv6 vrrp address 2 link-local fe80::{sprintf "%02x", $vln}:1
                    ipv6 vrrp address 2 global 300$blk:{sprintf "%02x", $vln}::1/64
                    ipv6 vrrp 2 enable
                    ip dhcp-relay
                    ip dhcp-relay fwd-path 1.1.1.1 mode dhcp
                    ip dhcp-relay fwd-path 1.1.1.1 enable
                exit
        @endfor
@endfor
```

In this example you can see sprintf() used to construct a string built from multiple ACLI variables.

In theory other Perl code can be used, as long as it returns some value when it has completed, though not all Perl code is guaranteed to work. The examples shown here are pretty much what the author has tested with. For additional requests contact the author.

SSH Integration

SSH known hosts

The ACLI terminal implements an SSH known hosts file, like any other SSH terminal. Note that the SSH server always supplies its public key, regardless of whether the SSH client then authenticates either via password or publickey authentication.

When connecting via SSH, the host SSH server provides its public key, and the SSH client (ACLI) looks into the known_hosts file to see if it can find a record for the host IP/hostname it is connecting to and whether a cached public key for that host exists. If an existing public key was cached in the know_hosts file and that key matches the key provided by the SSH server, then the SSH connection is made.

If instead no cached public key is found in the known_hosts file (or no konwn_hosts file exists yet) then the ACLI behaviour is determined by the *ssh_known_hosts_key_missing_val* ini key in the *acli.ini* file:

- **0** : SSH connection is refused
- **1** : User gets interactively prompted whether to add the key for the host in the known_hosts file, or to connect once without adding the key to known_hosts, or to abort the connection (this is the default behaviour)
- **2** : The key is automatically added to known_hosts file and a message is displayed to this effect (this used to be the default behaviour in ACLI versions up to 5.02 before this ini key was implemented)

The default behaviour is for ACLI to interactively prompt the user whether to trust the device and add its key to the known_hosts file, or connect once without adding the key to the known_hosts file or to abort the connection:

```
ACLI> ssh connect -l rwa 192.168.56.84

Logging to file: C:\Users\lstevens\Local-Documents\ACLI-logs\192.168.56.84.log
Escape character is '^]'.
Trying 192.168.56.84 .
acli-dev.pl: Host SSH key verification failed in known_hosts file, the key is missing!
acli-dev.pl: SSH Server key fingerprint is: ssh-rsa 2048 4a:f3:9d:13:a4:c4:bd:ab:17:0b:7b
acli-dev.pl: Press 'Y' to trust host and add key in known_hosts file
             Press 'O' to connect once without adding the key to known_hosts file
             Press any other key to abort the connection
Choice : Y

acli-dev.pl: Added SSH host key to known_hosts file
.
Enter Password:
.
Connected to 192.168.56.84 via SSH
acli-dev.pl: Performing login ............
acli-dev.pl: Detected an Extreme Networks device -> using terminal interactive mode
VSP-8284XSQ:1>enable
acli-dev.pl: Detecting device ...
acli-dev.pl: Detected VSP-8284-XSQ (00-51-00-91-f0-00) Single CPU system, 2 slots 84 port
acli-dev.pl: Use '^T' to toggle between interactive & transparent modes

VSP-8284XSQ:1#%
```

The importance of the known hosts file is to build trust of a given end system. The expectation is that whenever we connect to the same known host we receive the same key from that host. If this is not the case, then this can potentially indicate a compromised system or a man in the middle attack (though this usually also happens when the switch has been factory defaulted).

If a cached public key is found in the known_hosts file and the key does not match the key supplied by the SSH server, in this case the ACLI behaviour is determined by the *ssh_known_hosts_key_changed_val* ini key in the *acli.ini* file:

- **0** : SSH connection is refused (this used to be the default behaviour in ACLI versions up to 5.02 before this ini key was implemented)
- **1** : User gets interactively prompted whether to update the key for the host in the known_hosts file, or to connect once without updating the key in known_hosts, or to abort the connection (this is the default behaviour)
- **2** : The key is automatically updated with the new key in the known_hosts file and a message is displayed to this effect (Note, this is not a safe option)

The default behaviour is for ACLI to interactively prompt the user whether to trust the device and update its key in the known_hosts file, or connect once without updating the key in the known_hosts file or to abort the connection:

```
ACLI> ssh connect -l rwa 192.168.56.84

Logging to file: C:\Users\lstevens\Local-Documents\ACLI-logs\192.168.56.84.log
Escape character is '^]'.
Trying 192.168.56.84 .
acli-dev.pl: Host SSH key verification failed in known_hosts file, the key has changed!
acli-dev.pl: SSH Server key fingerprint is: ssh-rsa 2048 4a:f3:9d:13:a4:c4:bd:ab:17:0b:7b
acli-dev.pl: Press 'Y' to trust host and update key in known_hosts file
              Press 'O' to connect once without updating the key in known_hosts file
              Press any other key to abort the connection
Choice : Y

acli-dev.pl: Updated SSH host key in known_hosts file
.
Enter Password:
.
Connected to 192.168.56.84 via SSH
acli-dev.pl: Performing login ............
acli-dev.pl: Detected an Extreme Networks device -> using terminal interactive mode
VSP-8284XSQ:1>enable
acli-dev.pl: Detecting device ...
acli-dev.pl: Detected VSP-8284-XSQ (00-51-00-91-f0-00) Single CPU system, 2 slots 84 port
acli-dev.pl: Use '^T' to toggle between interactive & transparent modes

VSP-8284XSQ:1#%
```

Note that with the above default behaviours where the user is prompted to press 'Y' or 'O' or any other key, if the user takes too long to press a key, it is possible for the SSH connection to timeout, not because ACLI is timing it out, but because the SSH server side will not hold the socket open indefinitely. In this case, simply re-connect again.

If the default behaviour is changed to refuse the SSH connection (ini key set to **0**), then the only way to SSH connect to a host with missing or changed keys is to edit the SSH known hosts file and remove the existing entry and then reconnect. This can be done using the *'ssh known-hosts delete <ip>'* and *'reconnect'* commands under the ACLI control interface.

```
ACLI> ssh known-hosts ?
Syntax: ssh known-hosts [delete]

ACLI> ssh known-hosts delete ?
Syntax: ssh known-hosts delete <hostname/IP> [<tcp-port>]

 - Hostname or IP must exactly match entry in known_hosts file
ACLI>
```

The *known_hosts* file is looked for in the following paths in order:

- *%ACLI%\.ssh* (if you defined the *%ACLI%* path)
- *$HOME/.ssh* (on Unix systems)
- *%USERPROFILE%\.ssh* (on Windows)
- *%ACLIDIR%\.ssh* (ACLI install directory)

If a *known_hosts* file is not found, one will be created in the first existing path of the above.

Once connected via SSH, it is always possible to view the current SSH connection details via the embedded *'@ssh info'* command or the *'ssh info'* command under the ACLI control interface:

```
VSP-8284XSQ:1#% @ssh info
SSH Version 2
SSH Connected to 192.168.56.84
SSH authentication used : password
Server key fingerprint : ssh-rsa 2048 4a:f3:9d:13:a4:c4:bd:ab:17:0b:7b:39:f9:95:96:ad
SSH known_hosts lookup result : verified
```

SSH publickey authentication

To perform SSH publickey authenticayion the ACLI terminal needs to be pre-loaded with both the user's private and public keys. These keys need to be placed in any of the following directories:

- *%ACLI%\.ssh* (if you defined the *%ACLI%* path)
- *$HOME/.ssh* (on Unix systems)
- *%USERPROFILE%\.ssh* (on Windows)
- *%ACLIDIR%\.ssh* (ACLI install directory)

The expected filename is *'id_rsa'* or *'id_dsa'* respectively for RSA and DSA private keys; the corresponding public key (which is also required) is expected with filenames *'id_rsa.pub'* / *'id_dsa.pub'*. The ACLI command line *-k* switch can still be used to override the default filename keys.

The ACLI terminal SSH keys can be managed and inspected using either the embedded *'@ssh keys'* command or the *'ssh keys'* command under the ACLI control interface.

```
ACLI> ssh keys info
SSH keys loaded:
        SSH Private key           : C:\Users\lstevens\.ssh\id_rsa
        SSH Public key            : C:\Users\lstevens\.ssh\id_rsa.pub
        SSH key type              : RSA
        Passphrase encrypted      : No
        Data Encryption (DEK)     :
        Key length                : 2048 bits
        Key MD5 fingerprint       : 72:23:a1:8b:b1:0b:2f:fa:d5:0b:2b:b2:74:a5:f9:e7
        Key comment               : rsa-key-20160306
ACLI>
```

Note that the ACLI terminal does not supply any utility to generate publickeys. But you can generate your own keys using most other SSH terminals (Putty comes with a nice utility called Puttygen). Note that the ACLI terminal requires the SSH public/private keys to be in OpenSSH format.

Setting up SSH publickey authentication on Extreme devices

To use SSH publickey authentication it is not enough to load one's public+private keys on the ACLI client terminal side. It is then necessary to place one's public key on the SSH server (hosts). This is usually a fiddly process which varies across the different products.

For VOSS & PassportERS devices this process can be taken care of by ACLI. Simply connect to the target device (using either Telnet or SSH with password authentication) then issue the embedded commands *'@ssh device-keys'* to install your public key on the device.

```
VSP-8284XSQ:1#% @ssh device-keys install ?
Syntax: @ssh device-keys install admin|auditor|operator|privilege|ro|rw|rwa|rwl1|rwl2|rwl

VSP-8284XSQ:1#% @ssh device-keys install rwa
Installing SSH Public key on switch .......done!

VSP-8284XSQ:1#% @ssh device-keys list
Retrieving SSH Public keys on switch ....done!

File             Idx  Acc Levl  Format   Type     Bits  Fingerprint            Comment
---------------  ---  --------  -------  -------  ----  ---------------------  -------
rsa_key_rwa       1   rwa       ietf     ssh-rsa  2048                         rsa-key
                                         72:23:a1:8b:b1:0b:2f:fa:d5:0b:2b:b2:74:a5:f9:e7

VSP-8284XSQ:1#%
```

Once done, you can now connect to the device using SSH publickey authentication.

```
ACLI> ssh connect -l rwa 192.168.56.84

Logging to file: C:\Users\lstevens\Local-Documents\ACLI-logs\192.168.56.84.log
Escape character is '^]'.
Trying 192.168.56.84 .
Connected to 192.168.56.84 via SSH
acli-dev.pl: Performing login .............
acli-dev.pl: Detected an Extreme Networks device -> using terminal interactive mode
VSP-8284XSQ:1>enable
acli-dev.pl: Detecting device ...
acli-dev.pl: Detected VSP-8284-XSQ (00-51-00-91-f0-00) Single CPU system, 2 slots 84 port
acli-dev.pl: Use '^T' to toggle between interactive & transparent modes

VSP-8284XSQ:1#%
```

Notice that in the above SSH login, no password was set nor asked for by the switch.

To see how the SSH connection was performed, use the *'@ssh info'* embedded command or the *'ssh info'* command under the ACLI control interface:

```
VSP-8284XSQ:1#% @ssh info
SSH Version 2
SSH Connected to 192.168.56.84
SSH authentication used : publickey
Server key fingerprint : ssh-rsa 2048 4a:f3:9d:13:a4:c4:bd:ab:17:0b:7b:39:f9:95:96:ad
SSH known_hosts lookup result : verified
```

Keepalive Timer

It is usually a security requirement to configure a CLI timeout on the devices, so that stale sessions will automatically close. Regrettably the ACLI author is not having any of that and the ACLI terminal, in interactive mode, implements a keepalive timer which will never let sessions timeout. The keepalive timer can be set or managed under the ACLI control interface using the "terminal" command; by default it is set to 4 minutes, which means that at every expiry the terminal will automatically send a carriage return to the connected host. These carriage returns are sent in the background and are not visible on the ACLI session. The CLI timeout on the host device thus can never expire and the connection can be kept running by ACLI indefinitely, or to be more precise until ACLI's own session timer expires.

If it is desired to disable this ACLI feature (and let the devices timeout themselves) it is sufficient to set the keepalive timer to 0 under the ACLI control interface.

```
ACLI> terminal timers keepalive <timer in minutes; 0 = disable>
```

To always set the keepalive timer to zero (or to any other value) by default use the *acli.ini* file and set the *keepalive_timer_val* key. See the ACLI ini file section.

Session Timeout

If the ACLI keepalive timer is non null, then sessions will never time out. ACLI comes with a session timer which can be used to timeout connections but this is done on ACLI's terms, not the devices. The session timer can also be set or managed under the ACLI control interface using the *'terminal'* command.

```
ACLI> terminal timers session <timeout in minutes; 0 = disable>
```

By default it is set to 10 hours and can also be set to a different default value using the *acli.ini session_timeout_val* key. Setting the ACLI session timer to 0 will disable it, which, if the keepalive timer is non null, will result in the ACLI session never expiring (perhaps not a good idea...)

Tie-ing terminals together with sockets

Another major ACLI feature is the ability, from one ACLI session, to take control and tie to many other slave ACLI sessions, so that the user can configure multiple switches simultaneously. This is implemented by setting up UDP sockets on the loopback interface, using IP multicast, and is thus creatively named the socket functionality!

Other terminal programs offer a similar capability but in a much cruder form where all the slave terminals are character based and take every single character that the user types in the driving terminal. The ACLI socket functionality is much more sophisticated when used in interactive mode as it is command based and not character based. However the ACLI socket functionality is also available in transparent mode and in this case becomes equivelent to competing offerings.

The socket functionality is by default enabled in ACLI. Information about the feature can be viewed with the *'@socket info'* embedded command (or via *'socket info'* from the ACLI control interface).

```
VSP-8284XSQ:1#% @socket info

Socket settings:
        Socket functionality    : enable
        IP Multicast address    : 239.255.255.255
        IP TTL                  : 0
        Bind to IP interface    : 127.0.0.1
        Allowed source IPs      : 127.0.0.1
        Socket username         : lstevens
        Encode username         : enable
        Tied to socket          :
        Local Echo Mode         : error
        Listening to sockets    :
        Socket Name File        : C:\Users\lstevens\.acli\acli.sockets

VSP-8284XSQ:1#%
```

The above socket information is what is set by default. Most of these settings, like the IP addresses, echo mode and socket name file will never need changing (these will be discussed further on).

To get started it is sufficient to do just these two things:

1. On the slave ACLI sessions you will need to make the socket listen on at least one socket.
2. On the driving ACLI session you will need to "tie" to the socket of the slave terminals.

The sockets in question are UDP sockets, which are numbers. But numbers are not nice to use, so instead of referring to the socket by number the ACLI socket feature allows the use of socket names. On a fresh ACLI installation, there will be only one socket name defined, the implicit 'all' which also happens to set the base socket number for all other socket names created (by default set to UDP port 50000; this can be changed in the acli.ini file if necessary).

```
VSP-8284XSQ:1#% @socket names

Known sockets:
        all             50000
VSP-8284XSQ:1#%
```

To see how the sockets work, it is handy to have a couple (if not more) of ACLI sessions, and to merge both ACLI tabs into a single tab. Open both ACLI sessions in their respective tabs, then select the first tab, then right click on the other tab and select "Merge Horizontally"
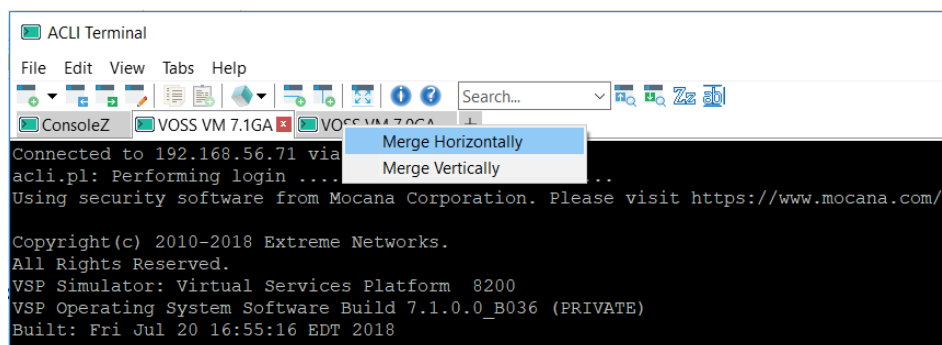
You should now have both ACLI sessions using the same tab and the window split in two horizontally:
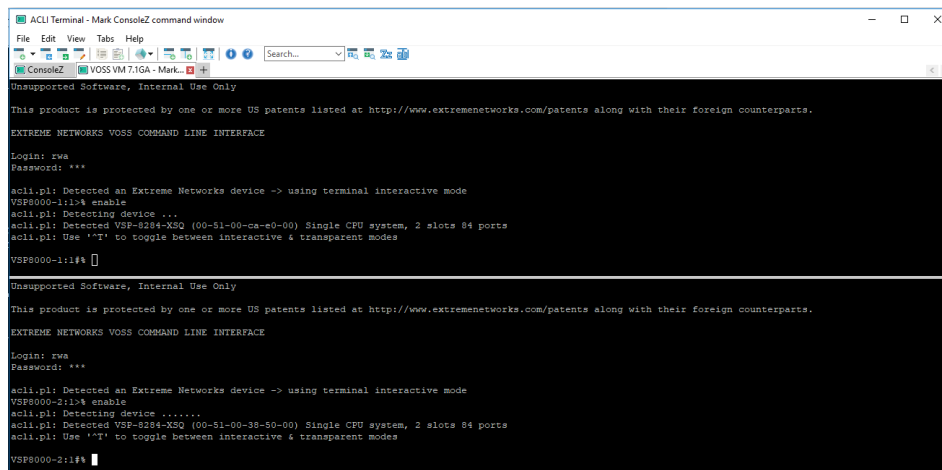
On the bottom session, we shall make the socket listen on the pre-defined socket name 'all':

```
VSP8000-2:1#% @socket listen
Listening on sockets: all

VSP8000-2:1#%
```

While on the top session, we shall tie the socket to the same pre-defined socket name 'all':

```
VSP8000-1:1#% @socket tie
Tied to socket 'all'

VSP8000-1:1#[all]%
```
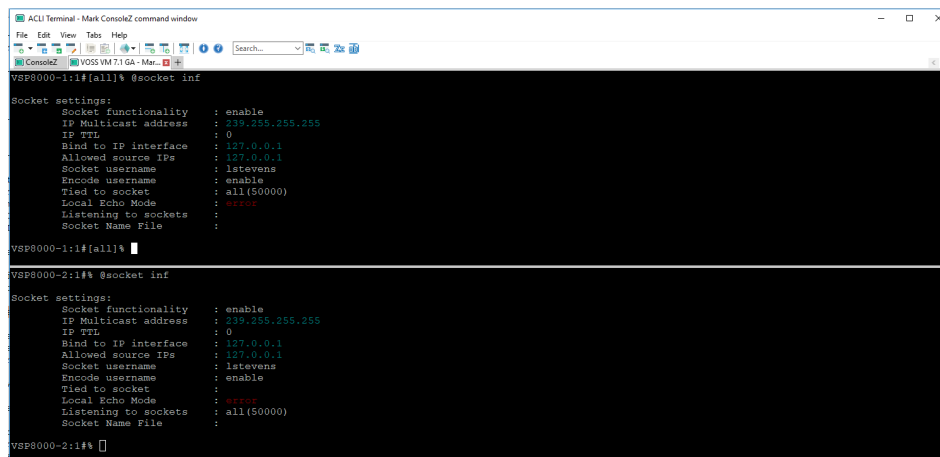
Notice that the prompt on the top session, which is where we tied the socket, now includes the tied socket name in square brackets *'[all]'*. This is useful, as it allows us to keep track of which socket an ACLI session is tied to and it also tells us which socket name.

Now, any command we enter on the top ACLI session, will get executed on the bottom session as well. In fact, the very moment the *'@socket tie'* was executed on the top session, a new CLI prompt

appeared on the bottom session. This is because whenever a socket is tied, the driving ACLI session will immediately send an empty command (a carriage return) to all other ACLI sessions listening on the same socket name. The reason for doing this is that typically you will not have the ACLI sessions in a split screen display as we have now, but instead you will probably have more than just two ACLI sessions and each session will be in its dedicated tab. A nice feature of the ConsoleZ window is that whenever new output is appended to a tab which is not selected, then the tab itself flashes. So simply by executing *'@socket tie'* on one ACLI tab, we can see which other ACLI tabs are flashing.

NOTE: If you want to just send a carriage return (and no command) to all listening terminals, without performing the *'@socket tie'* command again, simply hit the Return key multiple times in quick succession.

Now, it is not easy to show this in screnshots, but the *'@socket info'* command was only executed on the top ACLI session, yet we see it was executed in both sessions.

Note that the top session is tied to socket 'all' while the bottom session is listening to socket 'all'.

Only complete commands are sent to listening sockets in ACLI interactive mode. So in the driving tied ACLI session, expanding the command with tab or verifying the command syntax with '?' never need to be sent to listening terminals. Local more paging is also handled in such a way not to miss commands on listening sessions. If for example we empty the log file on the top ACLI session, with the terminal untied (so that we leave the log file untouched on the bottom ACLI session) and then dump the log file on both sessions from the tied top ACLI terminal, we'll end up with the bottom ACLI session left in the midst of dumping the log file, paused by a more prompt:

We could of course go to the bottom ACLI session and either hit 'Q' to quit more paging or hit space enough times to complete the output. However, if we were not interested in doing so and just wished to execute a new CLI command from the top, tied and driving, ACLI session, we can do so as the socket feature will automatically come out of any paused more prompt when it receives a new command on one of its listening sockets.

Notice that the "show users" command executed in the top ACLI session, automatically bumped the paused more prompt on the bottom ACLI session, which allowed the command to be correctly executed here as well.

In general, as long as ACLI is in interactive mode, whenever the driving ACLI terminal, which is tied to the socket, sends a command over the socket, the listening terminals will report back if they are not able to process a command. There can be many valid reasons why a listening terminal might not be able to process the command:

- The CLI connection to the switch might have been lost
- The listening terminal has been paused by entering ACLI control interface
- The listening terminal is already processing a command executed locally (unlikely if tied & listening terminals are running on same PC; but not otherwise)
- The listening terminal is not in interactive mode but in transparent mode
- In general, the listening terminal is not already locked on a CLI prompt when the first command is received from the socket

In the example above, the bottom listening ACLI session was first toggled into transparent mode by hitting CTRL-T, then the 'show users' command was executed on the top tied ACLI session. Note that we are alerted on the top session that a listening terminal is not able to process the command.
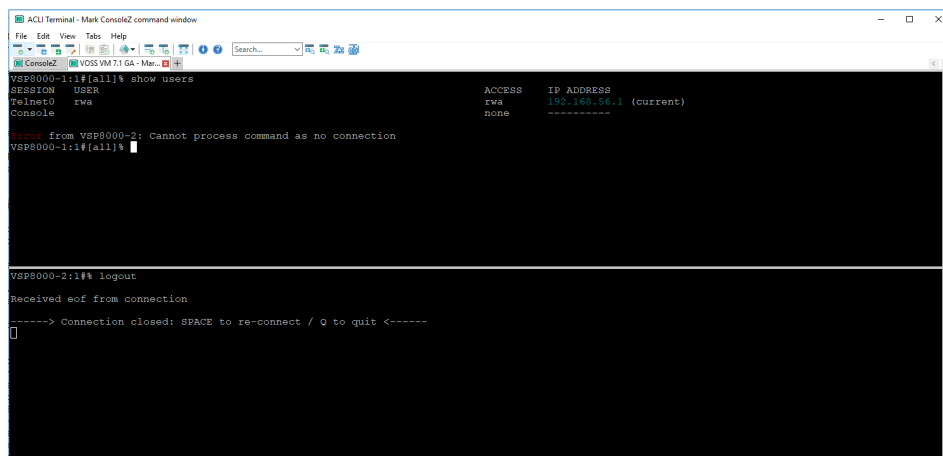
In the example above, first the ACLI control interface was invoked on the bottom listening ACLI session, then the 'show users' command was executed on the top tied ACLI session. Note that again we are alerted on the top session that a listening terminal is not able to process the command.

In the example above, a command which requires user input ('config', without 'term' in this case) was first executed on the bottom listening ACLI session without hitting return a second time; then the 'show users' command was executed on the top tied ACLI session. Note that again we are alerted on the top session that a listening terminal is not able to process the command. It would be inconvenient if the listening ACLI terminals were not in the expected state when we begin pushing commands to them via the socket feature.

Finally, in the example above, the connection of the bottom listening ACLI session was lost (in this case because we executed the 'logout' command on it); then the 'show users' command was executed on the top tied ACLI session. Note that again we are alerted on the top session that a listening terminal is not able to process the command because it has no connection.

It is worth also noting that all the above protections only apply to full commands executed on the driving tied ACLI session, only in interactive mode. However, even the interactive mode is capable of operating as a character based terminal (like in transparent mode). In fact, this is what happens during the time which follows execution of the last command and before we receive the next switch CLI prompt to lock onto. So if for example on the driving tied ACLI session we decide to execute 'edit config.cfg', this will enter a vi-like editor which is provided on VOSS platforms, and we will not get back a CLI prompt from the switch until when we exit such editing mode. During this time, the ACLI interactive terminal is in a "paced" sending mode, where it will operate as a character based terminal. Hence, if there were listening terminals which also executed the 'edit config.cfg' command, then any editing key strokes executed in the tied driving terminal will also get processed in the listening terminals. It is therefore possible to edit config.cfg files across many terminals simultaneously; however extra care needs to be taken in this context.

So, the basics of the socket functionlity have been covered in the above examples using just two ACLI sessions. However, if we had 15 (or more..) switches and we had set them all to socket listen on the 'all' socket name we could have configured them all at once from just one ACLI session. This is nice, but not always practical. Usually there will be groupings of switches which share a similar config (and which will all need updating if there is a need to change that config). These groupings will cut across the installed base in different ways. We might need to make changes to a pair of vIST switches; or we might need to make the change to every vIST switch in the network. We might want to make changes only on BEB nodes, or maybe on the BCB nodes, or maybe just nodes which are DVR Controllers. Or we might want to perform a software upgrade on all VSP4k switches, and later

on just the VSP7200 series ones. The idea of tie-ing to socket names is that multiple names can be defined for these different communities of switches.

Creating new socket names is as easy as simply configuring an ACLI session to listen to a new name. If the socket name is new, it will autatomticaly get allocated the next available socket number (e.g. 50001). If the socket name is not new, then it will use whatever socket number that name was allocated when it was first assigned by socket listen. All instances of ACLI, lookup socket names in the same resolution file and therefore will always use the same socket number for the same socket name. The actual syntax for listening to a socket name is the following:

```
@socket listen add <comma separated list of socket names or numbers>
```

In the examples above we used just *'@socket listen'* in its short form, which will add the pre-defined socket name 'all'. For any other socket name we need to use the full form.

So if we take our bottom ACLI session to VSP8000-2, this is a VSP8k, is a BEB and maybe is running DVR. So we might want it to listen to all these socket names (Note: names are arbitrary; user gets to choose the names which make most sense to him):

```
VSP8000-2:1#% @socket listen add v8,beb,dvr
Listening on sockets: all,beb,dvr,v8

VSP8000-2:1#% @socket info

Socket settings:
        Socket functionality    : enable
        IP Multicast address    : 239.255.255.255
        IP TTL                  : 0
        Bind to IP interface    : 127.0.0.1
        Allowed source IPs      : 127.0.0.1
        Socket username         : lstevens
        Encode username         : enable
        Tied to socket          :
        Local Echo Mode         : error
        Listening to sockets    : all(50000),beb(50034),dvr(50161),v8(50066)
        Socket Name File        : C:\Users\lstevens\.acli\acli.sockets

VSP8000-2:1#%
```

Notice that our ACLI session is now listening to 4 separate sockets. If on any other ACLI session a *'@socket tie'* is made on any of these socket names, then this ACLI session will be receiving commands from it. The listening socket names will also show the actual socket number, though this is not of much importance (and in your case the numbers may well get allocated differently).

So if we go back on our top ACLI session to VSP8000-1, we shall assume this is a BCB and a VSP8k, so we will let it listen to those socket names we allocated for these roles and we'll also add 'all' for good measure:

```
VSP8000-1:1#[all]% @socket listen add bcb,v8,all
Listening on sockets: all,bcb,v8

VSP8000-1:1#[all]% @socket info

Socket settings:
        Socket functionality    : enable
        IP Multicast address    : 239.255.255.255
```

```
                    IP TTL                    : 0
                    Bind to IP interface      : 127.0.0.1
                    Allowed source IPs        : 127.0.0.1
                    Socket username           : lstevens
                    Encode username           : enable
                    Tied to socket            : all(50000)
                    Local Echo Mode           : error
                    Listening to sockets      : all(50000),bcb(50208),v8(50066)
                    Socket Name File          : C:\Users\lstevens\.acli\acli.sockets

          VSP8000-1:1#[all]%
```

There a couple of things to notice here. For a start, the *'@socket listen'* command did not get executed on the bottom ACLI session, yet the top ACLI session is still tied to socket 'all' (as can be seen from the '[all]' on the VSP8000-1 prompt). Not all commands are sent to listening sockets, there are exceptions with some embedded commands. The *'@socket bind|echo|listen|ping|tie|untie'* and *'@acli'* embedded commands are never sent to listening ACLI sessions, otherwise these would make things messy!

The other thing to notice is that the top ACLI session is both still tied (to socket name 'all') and listening to all of socket names 'all', 'bcb' and 'v8'. An ACLI session can only be tied to one socket at a time, but it can be listening on more than one socket name simultaneously. Also an ACLI session which is tied to a socket name can also be listening on the same socket name as well as other socket names.

Let us now do some configuation work on our VSP8ks, and let us assume that we decide to do so from the bottom ACLI session to VSP8000-2, so we tie to the 'v8' socket name.

Notice that the top ACLI session, which was already tied to 'all', was also listening on socket name 'v8', which is now tied by the bottom ACLI session, therefore the 'all' tie of the top ACLI session was automaticaly bumped when *'@socket tie v8'* was executed on the bottom ACLI session.

Finally, to relase the socket tie on an ACLI session tied to a socket name simply use the '*@socket untie*' embedded command.

Whereas to stop listening on one or more socket names use one of the following:

- *@socket listen remove <comma separated list of sockets>* : Stops listening on specified sockets

- *@socket listen clear* : Stops listening on all sockets

Socket echo modes

The socket functionality comes with three possible echo modes. The socket echo mode determines what level of feedback it is desired to receive, on the tied terminal, from the listening terminals.

- **error** : The error echo mode is what has been demonstrated above and is the default echo mode. When a command is executed on the tied terminal, this is also sent to listening terminals and, if there was no error condition on those listening terminals with processing the command, then nothing is reported back to the tied terminal. If instead one (or more) of the listening terminals was either not able to execute the command or it did execute the command but the connected switch complained (generated an error) about the command, then the error condition is notified back to the driving tied terminal to alert the user; furthermore, if the user had pasted many commands into the tied driving terminal (i.e. in scripting mode) then the script will also halt.
- **all** : In this mode, the error reporting of the error mode also applies. However, in addition to that, all output generated by listening terminals is also sent back to the driving tied termnial. This mode is useful if one wants to execute a show command and visualize the output from all listening terminals on the driving tied terminal.
- **none** : If the error mode is set to none, then no reporting (error conditions) and no output is fed back to the tied terminal from listening ones. This is not a particularly useful mode.

The socket echo mode can be set globally with the following command:

```
@socket echo all|error|none
```

Follows an example using the 'all' echo mode:

```
VSP8000-1:1#% @socket echo all
VSP8000-1:1#% @socket tie
Tied to socket 'all'

VSP8000-1:1#[all]% cpu
      alias[all]% show khi performance cpu
    Slot:1
        Current utilization: 0
        5-minute average utilization: 0
        5-minute high water mark: 15 (09/01/18 13:33:08)

Output from VSP8000-2:
    Slot:1
        Current utilization: 0
        5-minute average utilization: 0
        5-minute high water mark: 9  (09/01/18 13:33:09)
VSP8000-1:1#[all]%
```

In this example we display the CPU utilization of the VOSS device connected to all the listening terminals and show all the outputs in the driving, tied, termnial.

Socket ping

When using the socket functionality to tie together many ACLI sessions, it can become hard to keep track whether all the listening sessions are in synch. Of course you can see the the ConsoleZ tabs flash when you tie the driving terminal to the socket name, and you get errors when executing commands if some listening sessions are not ready. Yet, a more proactive method is desireable.

The socket feature comes with a ping functionality which can make things a lot easier and allows the user to proactively check that all the listening ACLI sessions are ready to go, before even executing the first command. The socket ping can be executed with the following command:

```
@socket ping [<socket name>]
```

The socket name is optional, if executed on an already tied ACLI session. If not specified, the ping is performed on the same socket name to which the ACLI session is already tied to.

The socket name can be provided when it is desired to perform a socket ping on a different socket name from the one the session is already tied to, or to perform the socket ping on a non-tied ACLI session (i.e. the socket ping can be executed on listening terminals as well). On the ACLI session where the socket ping was executed, a response will be displayed for every ACLI terminal listening on the pinged socket name. A summary count of all responses is also displayed, so as to make it easier for the user to make sure all his sessions have responded.

In this example, *@socket ping* is executed without a socket name on an ACLI terminal which was already tied to socket name *v86*:

```
VSP8600-1:1#[v86]% @socket ping
Response from VSP8600-3
Response from VSP8600-2
Echo received from 2 terminals
VSP8600-1:1#[v86]%
```

In this example, *@socket ping* for socket *all* is executed on the same ACLI terminal which was already tied to socket name *v86*:

```
VSP8600-1:1#[v86]% @socket ping all
Response from VSP7200-2
Response from VSP8400-3 [in midst of --more-- output paging]
Response from ERS5900-STK
Response from VSP7200-1 [tied: v72]
Response from ERS4900-STK
Response from VSP8400-1 [tied: v84] [in midst of --more-- output paging]
Response from VSP7200-4
Response from VSP8400-3 [in midst of --more-- output paging]
Response from VSP8400-2 [in midst of --more-- output paging]
Response from VSP7200-3
Response from VSP8600-3
Response from VSP8200-1
Response from VSP8600-2
Echo received from 13 terminals
VSP8600-1:1#[v86]%
```

Note that we see a bunch of other terminals, and some info is provided on whether these terminals are already tied or if they are in the midsts of a more paging prompt.

```
VSP7200-4:1#% @socket ping v72
Response from VSP7200-2
Response from VSP7200-3
Response from VSP7200-1 [tied: v72]
Echo received from 3 terminals
VSP7200-4:1#%
```

In the above final example, *@socket ping* is performed on an ACLI session which is not tied.

Socket pre-defined aliases

ACLI ships with some pre-defined aliases for simplifying the socket functionality (NOTE: ACLI aliases are not limited to switch CLI commands, aliases can also be defined for ACLI embedded commands beginning with '@')

The first such alias is 'tie':

```
tie [<socket-name>] [<echo-mode: all|none>]
```

Let's try and execute this:

```
VSP8000-1:1#% tie v8
        alias% @socket tie v8; @socket echo error; @socket ping
VSP8000-1:1#% @socket tie v8
Tied to socket 'v8'

VSP8000-1:1#[v8]% @socket echo error
VSP8000-1:1#[v8]% @socket ping
Response from VSP8000-2
Echo received from 1 terminals
VSP8000-1:1#[v8]%
```

As can be seen above, the *'tie'* alias executes three separate embedded commands:

1. *@socket tie* : this ties the session (if no socket name was provided to the alias then the tie is to socket name 'all')
2. *@socket echo <mode>* : this sets the socket echo mode; if none was specified as argument to the 'tie' alias, then the more desireable default 'error' mode is always reset
3. *@socket ping* : after tie-ing to a socket, it always makes sense to execute a socket ping to make sure we have the expected listening terminal online

The next alias is *'listen'*:

```
listen [<socket-name>] [<if-string-match-against-name-or-model>]
```

Let's try and execute this:

```
VSP8000-2:1#% listen vist
        alias% @socket listen add vist
Listening on sockets: all,beb,dvr,v8,vist

VSP8000-2:1#%
```

As can be seen above, the *'listen'* alias is simply a short hand for executing the *'@socket listen add'* command.

The second optional argument (which can only be supplied if the socket name was provided) allows a tied terminal to execute the *'@socket listen add'* command on selected listening terminals. Earlier in this document it was said that some embedded commands cannot be sent over the socket functionality, in order not to make a mess. The *'@socket listen'* command is one of those commands. It just happens that the check is only performed over individual commands entered in the tied driving ACLI terminal. If the alias dereferences to a string with multiple commands, concatenated

with semi-colon ';' then the checks don't kick in, and it is possible to send these otherwise non-authorized embedded commands to the listening terminals. The *'listen'* alias takes advantage of this and embeds the *'@socket listen add'* in a little ACLI script which is sent as is to listening terminal to execute. The script allows listening terminals to verify whether their own CLI prompt or the switch model type of the connected host (e.g. VSP-8284-XSQ), matches the string which was provided by the user; only if there is a match, is the *'@socket listen add'* command executed by listening terminals.
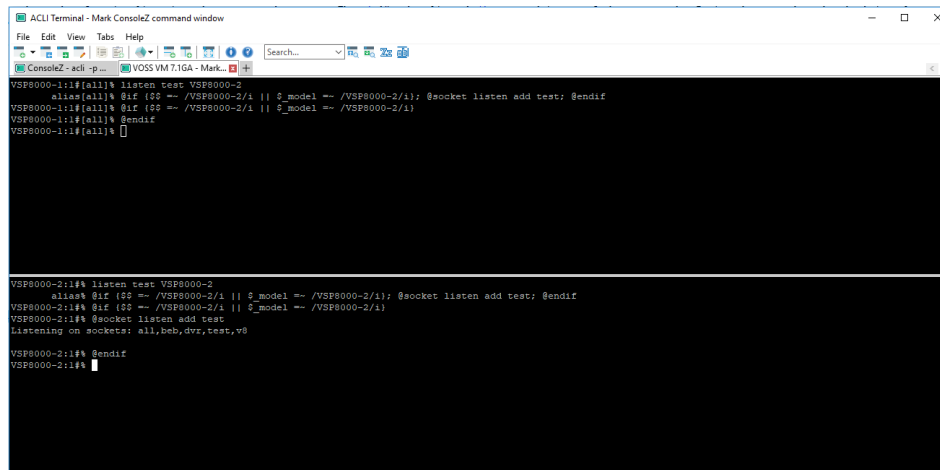
```
VSP8000-1:1#[all]% @alias show listen

listen [$socknames] [$condition]
    IF $condition eq ''
        THEN:
            @socket listen [add $socknames]

    ELSE:
            @if {$$ =~ /$condition/i || $_model =~ /$condition/i}
            @socket listen add $socknames
            @endif


VSP8000-1:1#[all]%
```

The ACLI author uses this, via a terminal tied to 'all', to conditional setup listening socket names on all other terminal (already listening on 'all'). See the following example:

Notice that the *'listen'* alias was invoked on the top ACLI session, and the alias dereferenced script was executed on both ACLI sessions. However the script ends up only executing the *'@socket listen add'* command on sessions where either the prompt ($$) matches the string provided or the switch model matches. In this example the *'@socket listen add'* only gets executed on the bottom, listening, ACLI terminal.

To complete the socket aliases, we also have 'sping':

```
sping [<socket-name>]
```

Which does:

```
VSP8000-2:1#% sping v8
        alias% @socket ping v8
Response from VSP8000-1 [tied: v8]
Echo received from 1 terminals
VSP8000-2:1#%
```

While 'untie' does a *'@socket untie'*:

```
VSP8000-1:1#[v8]% untie
        alias[v8]% @socket untie
VSP8000-1:1#%
```

And 'ignore' does *'@socket listen remove'*:

```
ignore [<socket-name>] [<if-string-match-against-name-or-model>]
```

Which does:

```
VSP8000-2:1#% ignore vist
        alias% @socket listen remove vist
Listening on sockets: all,beb,dvr,v8

VSP8000-2:1#%
```

The *'ignore'* alias also features the optional *<if-string-match-against-name-or-model>* argument as already discussed for the *'listen'* alias.

Socket send

In all the socket examples used so far, a driving ACLI session is tied to a socket and then commands are executed on this terminal and also executed on listening terminals. However, in some cases (usually in conjunction with ACLI scripting) it might be desireable to execute a command against a socket name (and hence against any listening ACLI termnials) without executing that same command on the tied terminal. For these use cases the socket functionality includes the *'@socket send'* command:

```
@socket send <socket name> [<wait-time-secs>] <command>
```

The first argument has to be the socket name. The optional wait time in seconds is useful if the socket echo mode is set to 'all' and some extra time is needed to retrieve the command output. The rest of the arguments are treated as the commnand to send. Note, use quotes if the command has leading or trailing space characters which need to be preserved. Follows an example:
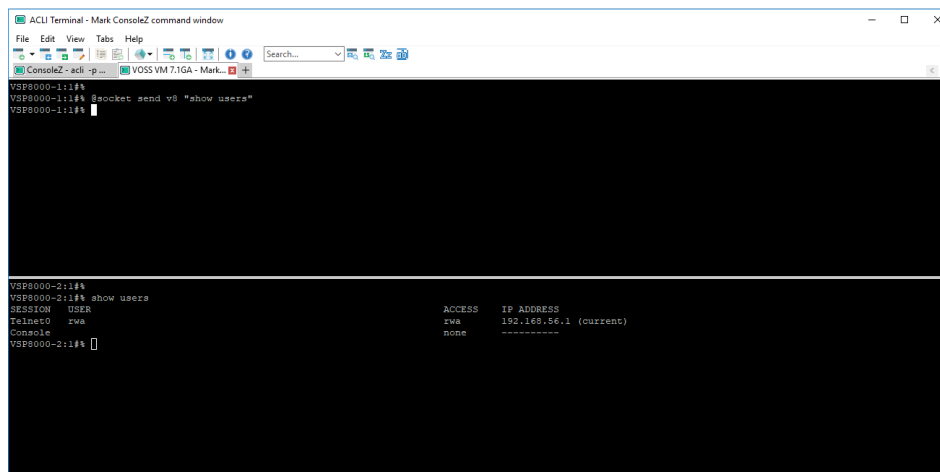
Note that the *'@socket send'* command was executed on the top ACLI session, but the *'show users'* command was only executed on the bottom ACLI session, which happens to be listening on the 'v8' socket name.

Socket names

As already mentioned in the previous socket sections, socket names come into existence the very moment an ACLI terminal is instructed to listen on a new name. When this happens the never used before socket name gets allocated the next available UDP socket number (starting from whatever number is set for the pre-defined *'all'* name - by default that number being 50000, if not changed in the *acli.ini* file). The new socket name is then recorded to the socket names file, which is always placed in path *%USERPROFILE%\.acli\acli.sockets*

This can always be found (if already accessed) by looking at the output of either the embedded *'@socket info'* command or the equivalent *'socket info'* command under ACLI control interface:

```
VSP8000-1:1#% @socket info

Socket settings:
        Socket functionality   : enable
        IP for sockets         : 127.255.255.255
        Socket username        : lstevens
        Encode username        : enable
        Tied to socket         :
        Local Echo Mode        : error
        Listening to sockets   : all(50000),bcb(50208),test(50018),v8(50066)
        Socket Name File       : C:\Users\lstevens\.acli\acli.sockets
        Bind to IP interface   : 127.0.0.1
        Allowed source IPs     : 127.0.0.1

VSP8000-1:1#%
```

Note that write access to the socket names file is protected via a lock, so only one ACLI instance is allowed to modify the file at any given time (performing a *'@socket listen add'* simultaneously on two or more ACLI instances for a brand new socket name can result in the operation temporarily failing on some instances).

The already defined socket names can be viewed either via inspection of that file or by using the *'@socket names'* embedded command:

```
@socket names [numbers]
```

Without the *'numbers'* argument an alphabetical list of the socket names is produced. With the *'numbers'* argument the list of socket names is by numerical order of the UDP port number. Follows the socket names which the ACLI author has used over time:

```
VSP8000-1:1#% @socket names

Known sockets:
        100g            50175
        1k              50082
        2nd             50005
        3k              50122
        3rd             50004
        4k              50020
        4kedge          50076
        4kist           50092
        5k              50049
        5k6             50089
```

```
7k          50021
7k12        50117
7k34        50197
7kb         50190
7kdca       50080
7ks         50191
7ks1        50192
7ks2        50193
7kxt        50065
8core       50007
8edge       50011
8k          50012
8kdist      50059
8ko         50061
9core       50006
9edge       50009
9k          50013
9kdut       50077
9kpoc       50023
acc         50086
accdist     50038
acepen      50010
al          50054
alag        50229
alh         50207
alh1        50228
all         50000
anex        50051
annex       50052
apls        50187
arm         50195
auh         50168
auhc        50171
auhl        50172
bcb         50208
beb         50034
bebe        50163
bgp         50039
blue        50196
bsedge      50068
ce          50180
cfm         50119
ciena       50072
cisco       50083
clh         50205
clh1        50227
client      50042
clp         50108
cluster     50053
cmp         50112
core        50036
dc          50041
dc1         50114
dc2         50113
dcbeb       50107
dhcp        50063
dist        50194
dlh         50206
dta         50165
dtb         50166
dtb4        50150
dtor        50149
dvr         50161
```

```
dvr4l      50160
dvrc       50152
dvrl       50153
dvrl1      50243
dvrl2      50244
dxb        50167
dxbc       50169
dxbl       50170
e3         50100
e4         50099
e4s        50102
edge       50050
ers        50177
esx        50069
f5         50016
fa         50067
fac        50132
fas        50174
fc         50179
fe         50136
felab      50154
fex        50109
fnnc       50140
fnne       50141
fnnf       50139
fw         50087
fwospf     50225
fwstat     50224
gns3       50199
ha         50048
hack       50138
ipmc       50044
ipv6       50091
isis       50002
ist        50056
isw        50235
ixia       50176
jas7k      50033
jas8k      50031
jasedge    50032
lab        50022
lacp       50026
lag        50226
lall       50158
ldvrl      50156
legacy     50151
lers       50173
ludo       50145
ludolab    50030
lv4        50157
lv8        50159
lvcd       50201
lvist      50188
lvsp       50162
ma         50178
macsec     50078
martin     50064
mce        50181
md         50182
mdfc       50183
mfc        50088
mpls       50040
msdp       50148
```

```
msec        50135
mux         50074
myname      50123
nlb         50015
ona         50185
oob         50084
oob3k       50236
oob4k       50237
ospf        50001
ospf2       50003
ous         50203
pair        50058
pall        50204
patch       50070
peer        50164
pim         50046
pimedge     50047
poc         50025
pod         50071
ppv8        50008
qos         50024
ring        50137
rob         50130
rv4         50200
salh        50219
sall        50216
san         50233
sb          50127
sbcb        50215
sclh        50214
script      50198
sdlh        50218
serial      50055
sers        50213
setup       50231
sfw         50223
show        50186
sim         50057
sl          50155
sln         50184
smlt        50014
sospf       50222
sp          50189
spb         50045
srv         50234
srvdist     50037
srvedge     50019
ss          50128
sspb        50217
ssw         50144
st          50062
stk         50081
stk2        50118
stks        50085
stkspb      50111
stp         50143
svist       50220
svrf        50221
swb         50142
tacacs      50230
temp        50035
test        50018
test1       50017
```

```
        three           50126
        tmp             50093
        tor             50060
        torbor          50043
        toto            50131
        two             50125
        ubp             50095
        v12             50146
        v23             50147
        v4              50096
        v412            50098
        v434            50103
        v44             50211
        v4412           50245
        v456            50101
        v4567           50105
        v48             50212
        v4d             50097
        v4l             50124
        v7              50115
        v7-12           50028
        v7-34           50027
        v7-56           50029
        v72             50241
        v7212           50242
        v8              50066
        v82             50120
        v84             50121
        v8412           50239
        v86             50210
        v8612           50238
        v8l             50129
        v9              50116
        v912            50133
        v934            50134
        vist            50106
        voss            50104
        vrf             50209
        vrp             50090
        vrrp            50094
        vsp             50079
        vxg             50232
        wan             50075
        wc              50110
        x69             50240
        xos             50202
        xt              50073

    VSP8000-1:1#%
```

Under the ACLI control interface, some additional commands are available to manage the socket names file:

```
    ACLI> socket names ?
    Syntax: socket names [clear|numbers|reload]
```

The *'clear'* argument will clear all user defined socket names. The *'reload'* argumet reloads the socket names file in memory; this would only be necessary if the *acli.sockets* file had been modified while ACLI instances were running.

Socket numbers

In the above sections sockets are always entered as names and ACLI automatically allocated a UDP port number (by default sequentially from port 50000) to the provided socket name(s). It is possible to also request a socket number, rather than a socket name. In versions of ACLI up to 5.07 it was sufficient to make the socket name a number and that number would be used as the UDP socket as well. So for example doing *"listen 5520"* would actually result in UDP port 5520 being used rather than a dynamically allocated port. This could result in undesireable consequences if for example ACLI was made to tie/listen on lower numbered ports reserved for other applications.

Hence, starting in ACLI vertsion 5.08 any socket name entered as a number will also be treated as a string and get allocated a dynamic UDP port number. In order to force ACLI to actually open the socket on a user specified UDP port number it is now necessary to use the following syntax, where the number provided is preceded by the "%" character:

```
VSP-8284XSQ:1#% @socket tie %20000
Tied to socket '%20000'

VSP-8284XSQ:1#[%20000]% @socket listen add %20001
Listening on sockets: %20001,all

VSP-8284XSQ:1#[%20000]% @socket inf

Socket settings:
        Socket functionality   : enable
        IP Multicast address   : 239.255.255.255
        IP TTL                 : 0
        Bind to IP interface   : 127.0.0.1
        Allowed source IPs     : 127.0.0.1
        Socket username        : lstevens
        Encode username        : enable
        Tied to socket         : %20000(20000)
        Local Echo Mode        : error
        Listening to sockets   : %20001(20001),all(50000)
        Socket Name File       :
```

Socket IP and username settings

The socket functionality is based on UDP sockets and, since the typical use case for using this functionality is where all ACLI instances are running on the same machine, the IP addresses used are always the system loopbacks. Furthermore, if using ACLI on a shared machine, by different users, it becomes necessary to encode the usernames into the messages sent over these socket otherwise one user might end up sending commands to ACLI sessions from a different user on the same machine. If looking at the output of *'@socket info'*:

```
VSP8000-1:1#% @socket info

Socket settings:
        Socket functionality    : enable
        IP Multicast address    : 239.255.255.255
        IP TTL                  : 0
        Bind to IP interface    : 127.0.0.1
        Allowed source IPs      : 127.0.0.1
        Socket username         : lstevens
        Encode username         : enable
        Tied to socket          :
        Local Echo Mode         : error
        Listening to sockets    : all(50000),bcb(50208),v8(50066)
        Socket Name File        : C:\Users\lstevens\.acli\acli.sockets

VSP8000-1:1#%
```

We can see that, by default:

- All sockets are bound to the loopback interface: 127.0.0.1
- When a command is received by a listening socket, the source IP is inspected and the command is only processed if it was received from an IP listed under allowed source IPs; this list by default only has the loopback IP: 127.0.0.1
- The destination IP used by a socket tied terminal to send socket messages is by default the IP multicast address: 239.255.255.255 (Older versions of ACLI used to use the loopback broadcast address 127.255.255.255 but this was changed as it only worked with Microsoft Windows and not with Unix based platforms)
- The IP TTL is set to 0 as we are only using the internal loopback interface by default
- The username is encoded in all messages sent over these sockets, using the user's username

The username is obtained autimatically by ACLI, only the encoding of it in socket messages can be enabled or disabled.

So, by default the ACLI socket functionality is pretty much locked down to operate only on the internal loopback address and for a single user.

However, technically, any IP address could be used, and the ACLI implementation allows the user to go and change those default IP addresses. So one could reconfigure the ACLI socket functionality to take control of ACLI instances running on a different machine across the network using IP multicast. This does work and is amusing but not particularly useful. If anyone wants to play with this, be aware that the socket name to UDP port mappings may not be the same on different machined for different users. In this case specify socket UDP port numbers (preceded by "%", as explained in previous section) instead of names.

Scripting

As explained in the Interactive mode section of this manual, the ACLI terminal, in interactive mode, operates in the fashion of a command based terminal (i.e. it is not character based, like other terminals, or indeed transparent mode). This means that ACLI, in interactive mode, has the ability to lock onto the device's CLI prompt, and henceforth execute discrete commands only. This architecture comes in very handy for writing CLI scripts, which can pretty much be as simple as a bunch of CLI commands.

Within the interactive mode, ACLI actually can operate in two sub-modes:

- **User input mode**: In this mode, the user types commands from STDIN (terminal keyboard) and only when the user hits Carriage Return does the ACLI terminal send the appropariate command to the host device. Then ACLI waits to lock onto the next CLI prompt to lock onto (in the interim, the terminal is in paced-sending mode, which is character based, and allows user to interact with the device in the event no CLI prompt is seen).
- **Scripting mode**: In this mode, the ACLI terminal has cached multiple commands to push to the host device, and will proceed, without any user intervention, to send each command as soon as it can lock onto the next CLI prompt.

In this chapter we shall only cover the scripting mode.

There are actually several ways which will result in ACLI entering scripting mode:

1. User copy-pastes a bunch of commands into the ACLI terminal
2. User executes a script file, which is an external text file containing the CLI script commands
3. User executes a run script
4. User executes a repeating command (using either the $@<interval$-$seconds>$ or $\&<values$-$list$-or-$range>$ notations)
5. User executes a semi-colon fragmented command (i.e. multiple commands on same line, separated by ';')

(4) and (5) are already covered in separate sections of this manual.

Pasting commands into the ACLI session

This is the easiest way to get a script going. Simply copy-paste your script or bunch of CLI commands into the ACLI terminal. The ACLI terminal is capable of making a difference between a user typing characters at the input keyboard and him pasting a whole bunch of text. In the latter case, the pasted text is immediately read into a script buffer, where it is chopped up into individual commands (the carraige return character identifies the end of a command) and is then processed as a script, one command at a time, and each command will wait in turn for the next device's CLI prompt. This is in stark contrast to what happens when you copy-paste multiple commands into a traditional character based terminal, which will simply regurgitate the entire text towards the host device in one shot; one is then at the mercy of the limited receive buffers of the host device, which if overrun will result in only a portion of the pasted text getting processed; the result is also usually messy as it becomes hard to follow how the connected device processed the commands and becomes hairy if one of the early commands generated an error!

With the ACLI approach, there is no limit to the size of the CLI script which is pasted into the session. A script of several hundred lines or more can be safely copy pasted into an interactive ACLI session.

Sourcing a script file

This is a more conventional approach, in that the user has prepared an offline text file which contains the CLI script. The ACLI embedded *@source* command can be used to load up the contents of that file and place it directly into the script buffer. There are a number of possible syntaxes:

```
@source <filename> [$1, $2, ...args]
@source <.ext> [$1, $2, ...args]
[<CLI command>] < <filename> [$1, $2, ...args]
[<CLI command>] < <.ext> [$1, $2, ...args]
```

If the script filename provided is just a file name with no path, the file will be searched for in the ACLI's working directory (which can be inspected with *@pwd* and changed with *@cd* embedded commands). Alternatively the script file can be specified with its path name and can then be located anywhere on the local machine file system.

In the second and last syntax forms, if only a dot extension is provided, e.g. *'.src'* then the filename which will be searched for will be *<switch-name>*. The last two syntax forms allow the use of the *'<'* character and can be placed directly after a CLI command. Not hugely useful, but syntactically allows the following:

```
config term < myscript.src
```

What happens here, is that first the *'config term'* CLI command is executed and then the script takes over from there.

Optional arguments can be provided on the command line, and these arguments will then be available to the ACLI script, once running, by simply invoking variables:

- **$<number>** : $1 will hold first argument; $2 the second argument; and so on...; arguments are separated by white space characters or enclosed in quotes
- **$*** : This variable will hold a string containing all the text entered as variables

Run scripts

Run scripts are a more convenient way of sourcing a script file, within a nicer framework. A run script is executed via the *@run* embedded command:

```
@run <runscript> [$1, $2, ...args]
```

The syntax for *@run* is identical to the syntax for *@source* in that the file can be a filename with or without a path and the optional arguments are processed in exactly the same way as for the *@source* command.

The first difference of a run script is that if the *<runscript>* filename has no path, then ACLI will not look for the run script in the current working directory, but instead will look for the script in the following paths, in the following order:

- ENV path *%ACLI%* (if you defined it)
- ENV path *$HOME/.acli* (on Unix systems)
- ENV path *%USERPROFILE%\.acli* (on Windows)
- ENV path *%ACLIDIR%* (the ACLI install directory)

The paths under which run scripts will be looked for can be viewed using the *@run path* embedded command:

```
PSEUDO#% @run path

Paths for Run Scripts:

    Origin    Path
    ------    ----
    private   C:\Users\lstevens\.acli
    package   C:\Users\lstevens\Scripts\acli

PSEUDO#%
```

The second difference is that if a filename with no extension was provided, then ACLI will look for a file with the *'.run'* extension, since this is the conventional extension for ACLI run scripts.

The third difference is that ACLI maintains an easy list of available run scripts and performs version reporting against them. This informaton can be easily viewed using the *'@run list'* embedded command:

```
PSEUDO#% @run list
Available Run Scripts:

    Name         Origin    Vers   Description
    ----         ------    ----   -----------
    2500fw       private   1.00   Manages ERS2500 ACL FW rules in Ludo's HomeLab
    atf-pwd      private   1.00   Change device passwords, for demo events
    ctc-mgmt     private   1.02   Setup SNMP, Syslog & SSH on the switch for Valbonne POC s
    ers-default  private    n/a   Create a rwa/rwa account to match Modulars
    fe           package   1.00   Setup Fabric Extend on the switch
    patch        private   1.00   Patches clients in Ludo's HomeLab
    poc-nms      private   1.01   Setup SNMP, Syslog & SSH on the switch for Valbonne POC s
    poc-snmp     private   1.01   Setup SNMP on the switch for Valbonne POC setup
    poc-ssh      private   1.00   Setup SSH on the switch for Valbonne POC setup
    snmp         package   1.03   Setup SNMP on the switch
    test         private    n/a   Sandbox test run script

PSEUDO#%
```

Run scripts are categorized as either:

- **Package**: These are run scripts located under the ACLI install directory; typically these are run scripts which are shipped with ACLI, and can be updated whenever the ACLI update script is executed, if a more recent version of the script is available.

- **Private**: These are run scripts located in the user's private path (either *%USERPROFILE%\.acli* or *$HOME/.acli* or *%ACLI%* if it was defined); this is where the user should place his or her run scripts.

The *'@run list'* command also extracts the script version, if included (script should simply contain a line like *'# Version = <version>'*), and a description of the script (this needs to be the first commented line inside the run script file). If you have many run scripts, this becomes a neat way for keeping track of them all!

Note: given the *@run* command syntax, there can never be a run script called either *'list'* or *'path'* !:

```
VSP-8284XSQ:1#% @run ?
Syntax: @run  [$1, $2, ...args]
        @run list|path

VSP-8284XSQ:1#% @run
```

Currently the *snmp* run script is the only package script distributed by ACLI; it is provided more as a showcase of how ACLI scripting can be used:

```
VSP-8284XSQ:1(config)#% @run snmp

VSP-8284XSQ:1(config)#% @echo off output off

This script will help you setup SNMP on the switch
Hit Enter to continue (CTRL-C to quit) :
```

Note that we execute *'@run snmp'* but the actual script filename is *snmp.run*.

Error detection

A common challenge when writing CLI scripts is to get the script right. Often this is a trial and error process, to get the right switch CLI commands, with the right syntax and in the right order. What usually happens is that somewhere in the middle of the CLI script, the host device throws an error on a CLI command, and does not execute it. If the CLI script does not detect that error, and continues to send subsequent CLI commands, the chances are that most subsequent commands will also produce errors and once the script has happily finished sending all the CLI commands, the user is left with a tangled mess of error messages and no clue which commands correctly executed and which commands did not. Recovering from these situation is painful and time consuming.

ACLI scripting solves this problem, because it is capable of detecting error messages from the connected device. An ACLI script sends one command at a time, and waits for the next CLI prompt before sending the next CLI command. If an error message is detected from the connected device, then ACLI automatically halts script execution. This is ideal, since the user can easily inspect the command which was not accepted and make the necessary alterations, without having to work through a trail of error messages. As an example, consider the following CLI script to create a VLAN on a VOSS device:

```
config term
vlan create 666 type port-rstp 0
vlan members add 666 1/6
ifv 666
   ip address 10.10.10.10/24
exit
end
```

There is an intentional error in the vlan create command. If we run that script in ACLI we get the following:

```
VSP-8284XSQ:1#% config term
Enter configuration commands, one per line.  End with CNTL/Z.
VSP-8284XSQ:1(config)#% vlan create 666 type port-rstp 0
                                                         ^
% Invalid input detected at '^' marker.
VSP-8284XSQ:1(config)#%
```

Notice that the script stops at the command where an error was received.

Even better, once the problem is resolved, the user can even chose to resume the script execution from where it left using the *@resume* embedded command (this is covered in the 'Resume buffer' section below)

Error detection is actually configurable. By default it is enabled, but the user, or indeed the script itself, can decide to disable error detection using the *@error enable|disable* embedded command.

```
VSP-8284XSQ:1#% @error ?
Syntax: @error disable|enable|info|level

VSP-8284XSQ:1#% @error info

Host error detection        : enable
Host error level detection : error

VSP-8284XSQ:1#%
```

If error detection is disabled, then ACLI will not halt script execution if the connected device produces an error message, and will keep running. There are reasons where it is useful to disable error detection:

1. The ACLI script is just a bunch of show commands, to collect switch data and debug information for offline processing. On the VOSS platforms, there is a 'show full-tech' command which attempts to dump a number of switch show commands, but often some of these dumps includes an error message. In this case it is not useful for the ACLI script to stop execution, so the CLI script simply needs to start with a *'@error disable'* statement.

2. Sometimes, the script needs to check whether or not a certain CLI command exists on the connected device, and to act accordingly. In this case, having the script halt if the CLI command is not accepted, is not useful.So the ACLI script can itself disable error detection temporarily by including the *@error disable|enable* commands before and after trying out the CLI command. In this use case, the *$@* reserved variable can be used to check if a command did in fact produce an error message or not.

The following example illustrates point (2) above:

```
#
# Create on ERS stackable an rwa/rwa account to match VOSS VSPs
#
@print "Creating rwa/rwa user ..."
no password security
@error disable
        username add rwa role-name RW password // rwa // rwa
@error enable
@if $@
        @print "WARNING: Old switch software, cannot create rwa user"
@endif
# Script execution continues in either case..
```

The script sequence tries to create an rwa/rwa user account on an ERS stackable switch. The *'username add'* CLI command is supported on recent ERS models, but not on older ERS models. We could have written the script to verify the ERS model, and work out on which models the CLI command is present and on which models it is not, but in this case it simply easier to try the command and see what happens. Because we don't want the script to halt execution in case the CLI command is not accepted, we sandwitch the CLI command in between an *@error disable* and *@error enable* lines. We then check the reserved variable *$@* (which will hold the actual error message of the last CLI command executed against the switch) and if it is set, then script notifies the user that the username could not be set. Either way, the script execution is not halted and the script continues execution.

ACLI scripting error detection can be set to two levels: error and warning:

```
@error level error|warning
```

- **error** : ACLI script execution will stop only at error messages. An error message indicates that the CLI command was not executed, for some reason.
- **warning**: ACLI script execution will stop at error messages but also warning messages. A warning message indicates an anomaly, but the CLI command which generated it did actually execute. Note, the warning level is not fully implemented and may not catch all warning messages.

User interaction during script execution

So an ACLI script will automatically halt execution if it encounters an error. There is however another way to halt script execution, and that is when the user interferes with the script execution. Imagine you've launched a large script, and while it is executing you hesitate and decide that no, you do not want the script to run and instead you want to stop it! To stop the script, it is sufficient to hit any key on the keyboard on the ACLI session which is runnnig the script. I.e. simply hit the Return key to halt the script.

The user can then reflect on the greater wisdom of his script, see how far it executed, and can also inspect the remaining portion of the script which has not executed via the *'@resume buffer'* embedded command (this is covered in the 'Resume buffer' section below). And if the users decides that after all the script is safe to run, he can resume it from where it was halted simply by executing the *@resume* command!

Resume buffer

In the preceding sections it was already mentioned that when an ACLI script is halted, either because an error was encountered or because the user interfered with the script execution, its execution can be resumed.

Let us revisit the same example where the CLI script to create a VLAN is halted, because of an invalid command. The script:

```
config term
vlan create 666 type port-rstp 0
vlan members add 666 1/6
ifv 666
   ip address 10.10.10.10/24
exit
end
```

And it's execution:

```
VSP-8284XSQ:1#% config term
Enter configuration commands, one per line.  End with CNTL/Z.
VSP-8284XSQ:1(config)#% vlan create 666 type port-rstp 0
                                                          ^
% Invalid input detected at '^' marker.
VSP-8284XSQ:1(config)#%
```

The ACLI *@resume* embedded command has the following syntax:

```
@resume [buffer]
```

Issuing the command *'@resume buffer'* will dump the ACLI script buffer, where we can see the remaining part of the script which did not execute:

```
VSP-8284XSQ:1(config)#% @resume buffer

------ STDIN pasted buffer ------
vlan members add 666 1/6
ifv 666
   ip address 10.10.10.10/24
exit
end
------ End of resume buffer -----
VSP-8284XSQ:1(config)#%
```

The script buffer will also provide information on the origin of the script in the buffer. In this case the CLI script was simply copy-pasted into the ACLI session.

If we wanted to resume the script execution, after having corrected the *'vlan create'* command, we simply need to execute the *@resume* command:

```
VSP-8284XSQ:1(config)#% vlan create 666 type port-mstprstp 0
VSP-8284XSQ:1(config)#% @resume
VSP-8284XSQ:1(config)#% vlan members add 666 1/6
VSP-8284XSQ:1(config)#% ifv 666
                alias% interface vlan 666
VSP-8284XSQ:1(config-if)#%    ip address 10.10.10.10/24
VSP-8284XSQ:1(config-if)#% exit
VSP-8284XSQ:1(config)#% end
VSP-8284XSQ:1#%
```

Note that in the above output, the user only typed the commands on the first two lines. The rest was executed by the ACLI script once its execution had resumed.

Embedding scripts

ACLI scripting allows embedded scripts, provided the nested scripts come from a different source. A simple example:

```
config term
vlan create 666 type port 0
vlan members add 666 1/6
@sleep 2; ifv 666; ip address 10.10.10.10/24; exit
vlan create 777 type port 0
vlan members add 777 1/7
ifv 777; @sleep 2; ip address 77.77.77.10/24; exit
end
```

We simply copy paste the above CLI script into our ACLI session, and when script execution reaches the *'@sleep 2'* statement, we interrupt the script by hitting the carriage return key. We then inspect the script buffer via the *'@resume buffer'* command:

```
VSP-8284XSQ:1#% config term
Enter configuration commands, one per line.  End with CNTL/Z.
VSP-8284XSQ:1(config)#% vlan create 666 type port 0
VSP-8284XSQ:1(config)#% vlan members add 666 1/6
VSP-8284XSQ:1(config)#% @sleep 2; ifv 666; ip address 10.10.10.10/24; exit
VSP-8284XSQ:1(config)#% @sleep 2
VSP-8284XSQ:1(config)#%
VSP-8284XSQ:1(config)#%
VSP-8284XSQ:1(config)#% @resume buffer

------ Expanded cmd buffer ------
ifv 666
ip address 10.10.10.10/24
exit
------ STDIN pasted buffer ------
vlan create 777 type port 0
vlan members add 777 1/7
ifv 777; @sleep 2; ip address 77.77.77.10/24; exit
end
------ End of resume buffer -----
VSP-8284XSQ:1(config)#%
```

We see that when we hit the line which contains multiple commands which are smicolon ';' fragmented, a new script buffer is loaded with these commands and is inserted into the script execution.

The same is true when sourcing script files. For example a run script (or a sourced script) could contain a *@source* command. Which will result in a separate script file being loaded, and nested, within the first script file. There is no limit to the amount of script nesting which can be done this way. In theory this could allow a main ACLI script to run other child scripts as if they were subroutines. There is only one limitation, and that is that a sources script (or a run script) is not allowed to run itself recursively (otherwise this would end up in a infinite scripting loop, until memory is exhausted!)

Auto-responding to CLI command confirmation prompts

One of the challenges of scripting the CLI, is that some CLI commands ask for user confirmation. Usually with a prompt down the lines: "Are you sure ? (y/n):". If you are trying to script the CLI, this is very annoying as you'd have to check for such a prompt, and feed a *'y'* character to it, or else know in advance which commands are likely to produce such a confirmation prompt and insert an extra line in the CLI script, with a single *'y'* character followed by carriage return. For some of these commands, some switches allow the user to append a trailing -*y* to the command, and then the switch does not ask for confirmation. However, support for this is not consistent.

The ACLI terminal also implements the -*y* switch, which means that even if the switch does not have an option to supress the confirmation prompt, then ACLI can suppress it instead. In this case what actually happens is that if the switch sends the confirmation prompt, ACLI intercepts it and automatically feeds a *'y'* + carriage return to the switch. None of this is actually visible to the ACLI user, as ACLI does it in the background and the confirmation prompt is not printed on the ACLI session (see also chapter on 'CLI augmented switches'). So, using the -*y* switch is one way of solving the problem. However one has to remember to place the -*y* switch on the commands which need it.

So ACLI scripting makes things even more simple. Once in ACLI scripting mode, it is not necessary to place the -*y* switch on any CLI command. This is because, in scripting mode (and only in scripting mode), if ACLI does see a confirmation prompt, it will always automatically feed a *'y'* + carriage return to the switch. This achieves the desired behaviour when doing CLI scripting. There is one exception however. If the confirmation prompt contains either the *'reset'* or *'reboot'* keywords, then ACLI will not automatically respond yes and the confirmation prompt is not intercepted but is presented to the user; script execution will also pause, as no CLI prompt is received. This is because, if the switch resets, then the connection is lost, and in any case the ACLI script will halt. In this scenario the user would then have to manually enter *'y'* for the script to resume (and for the switch to reboot presumably..). If the intention is really for the ACLI script to actually reboot the switch, and one did not want the confirmation prompt to halt the script, then simply append -*y* to the reset command and in this case the script will carry ahead with the switch reboot command.

Feeding data to a CLI command's own prompts

Another challenge faced when scripting the CLI is that some commands interactively ask the user for some data input. A typical example is the CLI commands where the login or snmp credentials are set or changed. These commands usually do not allow the new passwords to be specified on the command line; instead they prompt the user to enter the old password, then ask the user to enter the new password, and then again to confirm the new password a second time. Scripting these commands becomes tricky.

Not with ACLI though. ACLI is able to detect if the last executed CLI command is asking for some user input, and ACLI allows the user to specify a CLI command upfront with all the information it might need. This is achieved with the following syntax, which allows any number of inputs to be fed to a given CLI command:

```
<CLI command> [-hf] // [<input>]
<CLI command> [-hf] // <input1> // <input2> ...
```

As an example, the following command creates a new user on ERS stackable platforms:

```
username add admin role-name RW password // P@ssw0rd // P@ssw0rd
```

In the background, the switch will prompt for the new admin password twice, but ACLI will automatically feed *'P@ssw0rd'* both times.

The optional *-h* or *-f* switches allow the input data to be cached for future invocation of the same command on the same device or same family type device respectively.

Note that the above syntax can also be used to make ACLI simply skip the prompt without providing any data.

```
<CLI command> //
```

For example, the following command:

```
VSP-8284XSQ:1#% config //
Configuring from terminal or network [terminal]?
Enter configuration commands, one per line.  End with CNTL/Z.
VSP-8284XSQ:1(config)#%
```

The above *'config //'* command achieved the same result as if we had entered *'config term'*.

Using Control structures with ACLI scripts

Creating CLI scripts can be as simple as just a list of CLI commands, which are sequentially executed against the connected switch. However, for more complex scripts, it is necessary to have some control structures.

ACLI offers a set of basic but essential control structures, in the form of the following embedded commands:

```
@if <cond>, @elsif <cond>, @else, @endif          if / elsif / else conditional operato
@while <cond>, @endloop                           while loop construct
@loop, @until <cond>                              loop until construct
@for <$var> &<start>..<end>[:<step>], @endfor     for loop construct using range input
@for <$var> &[']<comma-separated-list>, @endfor   for loop construct using list input (
@next [if <cond>]                                 jump to next value in a for loop cons
@last [if <cond>]                                 break out of a while, until or for lo
@exit [if <cond>]                                 break out of sourced script
@stop ["stop-message"]                            break out of sourced script and halts
```

The following script snippet shows how these structures can be used:

```
#
# Create DNS Name server
#
$name_servers = 20.9.190.160,20.9.190.159,20.9.190.158
$domain_name = reading.ctc.local
@print "Setting DNS name servers ..."
$_ = 1
@for $srv &$name_servers
        @if $_ eq 1
                ip name-server primary $srv
        @elsif $_ eq 2
                ip name-server secondary $srv
        @elsif $_ eq 3
                ip name-server tertiary $srv
        @endif
        $ = {$ + 1}
@endfor
ip domain-name $domain_name
```

All the ACLI control structures can be nested. There is no limit to the amount of nesting possible (or rather, the limit is the memory on your machine!).

Note that the above control structures are all implemented using ACLI embedded commands, but these commands will only work in ACLI scripting mode. So if you want to play with them, you will need to either copy-paste your script snippet into an ACLI session, or else place your script into a run file and execute it with *@run* or place it into any text file and execute it with *@source*.

The difference between *@exit* and *@stop* is that both will terminate the currently running script but if each script was executed by dereferencing an alias or a dictionary command, of which many were pasted or sourced from file, then *@exit* will not prevent the next alias or dictionary command from being processed, while *@stop* will halt all sourcing of those pasted/sourced commands, as if an error had been encountered. After *@stop* execution can be resumed with *@resume*.

Controlling ACLI output during script execution

ACLI scripts can become quite sofisticated. They can be made to prompt the user for input (see 'Querying user to set variable' in the 'Variables' section), they can be written with *if/elsif/else* control structures, they can perform *for* loops, *while* loops and *until* loops and they can easily execute any CLI command against the connected host device.

But sending many commands to the host device will still create a lot of output on the ACLI window which might be interesting for debugging the script initially, but becomes just undesireable output if the script executes well without any errors.

The idea of writing an ACLI script, in particular a run script, is to automate some series of CLI commnds, maybe perform configuration of some complex feature by simply asking the user for the necessary input and then executing the necessary commands. If the script is successful and all commands execute without any errors, it would be sufficient to show the user a single line saying that the configuration is successful, without printing out any of the commands executed and their output.

The ACLI terminal offers this capability via the *@echo* embedded command, which has the following syntax:

```
@echo off [output off|on]
@echo on
@echo sent
```

When a CLI command is executed, there are two parts of the output seen on the terminal window. There is the command itself, which is printed next to a CLI prompt, the output of the command (if it generated any output), and a new CLI prompt.

When running in scripting mode, it is possible to control whether or not an executed CLI command is to produce any output on the terminal screen. By default *@echo* is on, and any command will be displayed in full.

```
VSP-8284XSQ:1#% @echo info

Echo of commands & prompts : on
Echo of command output     : on

VSP-8284XSQ:1#%
```

If we execute *'@echo off output off'* in a script, then any subsequent CLI commands sent to the connected device by the ACLI script will not produce any output on the terminal window. Let us consider the following script:

```
@echo off output off
config term
vlan create 666 type port-mstprstp 0
vlan members add 666 1/6
ifv 666
   ip address 10.10.10.10/24
exit
end
@print "VLAN 666 was successfully created!"
```

If we copy-paste the above into an ACLI session (to a VOSS device) we will get the following output:

```
VSP-8284XSQ:1#% @echo off output off
VLAN 666 was successfully created!
VSP-8284XSQ:1#%
```

Notice that after the *'@echo off output off'* command, we do not see any of the output from the rest of the CLI commands in the script file. Yet these commands did execute, and the VLAN was created. Only the output of the embedded *@print* command is seen, as this output is never suppressed by the *'@echo off output off'* action.

If instead one wanted to display the output of some CLI command, but still hide the command itself and the prompt line, then one can use *'@echo off output on'*. Consider this script:

```
@echo off output on
@print "Thie time is:"
show clock
```

When pasted into an ACLI session we get:

```
VSP-8284XSQ:1#% @echo off output on
Thie time is:

Sun Sep 23 08:37:19 2018 UTC

VSP-8284XSQ:1#%
```

Notice that we get the output of the *'show clock'* CLI command, but we managed to suppress the CLI prompts and the *'show clock'* command from the output.

The *@echo sent* option is a variant where embedded commands are suppressed, but commands actually sent to the connected device are not. This mode was added for the ACLI dictionary functionality. See the *"Dictionaries"* section for more information.

By default *@echo* is enabled for both command & prompts and for the output. Simply executing *'@echo off'* will disable it just for the command & prompts, but not for the output; so in the above example we could have just used *'@echo off'*. Echo settings can be changed either in scripting mode or in non scripting mode; in non-scripting mode you will get a warning if you do so:

```
VSP-8284XSQ:1#% @echo off
Note: turning off echo only has an effect when sourcing/pasting commands

VSP-8284XSQ:1#% @echo info

Echo of commands & prompts : off
Echo of command output     : on

VSP-8284XSQ:1#%
```

Another important point to keep in mind, is if the echo modes are changed within the ACLI scripting mode, then when the script completes, the echo settings are automatically reset to on. That is not the case if the echo settings were set outside of scripting mode.

Hiding the CLI commands and their output is handy, except when there is a problem with the script and maybe the connected switch did not like one of the CLI commands. Then one would be left clueless as to what happened. To avoid such scenarios, the ACLI echo functionality will not suppress output on CLI commands which generate an error. Consider the following example:

```
@echo off output off
config term
vlan create 666 type port-mstprstp 0
vlan members add 666 1/6
ifv 777
   ip address 10.10.10.10/24
exit
end
@print "VLAN 666 was successfully created!"
```

There is an intentional mistake in the above script; the *'ifv'* alias command (which resolves to *'interface vlan'*) is pointing to the wrong vlan id (777 instead of 666). If we execute the above script we get:

```
VSP-8284XSQ:1#% @echo off output off

VSP-8284XSQ:1(config)#% ifv 777
                alias% interface vlan 777
```

```
      % Vlan 777 does not exist% Vlan 777 does not exist

      VSP-8284XSQ:1(config)#%
```

Notice that the *@echo* statement had disabled command & prompts as well as output; yet, because we hit an error on the *'ifv'* alias command, then we get to see the offending command and the error message which it generated. The script obviously halted at this very command. Inspection of the resume buffer confirms this:

```
      VSP-8284XSQ:1(config)#% @resume buffer

      ------ STDIN pasted buffer ------
         ip address 10.10.10.10/24
      exit
      end
      @print "VLAN 666 was successfully created!"
      ------ End of resume buffer -----
      VSP-8284XSQ:1(config)#%
```

So all is good!

Sending raw text to connected host

Sending a CLI command is as easy as simply including a CLI command line in the script file. However there may be a need to send raw text to the connected host, without necessarily expecting a new prompt. In this case, if no prompt is seen coming back, the ACLI script will not continue execution. For these scenarios, the embedded *@send* command can be used to send raw text or raw characters to the connected host.

```
@send brk|char|ctrl|line|string

@send brk
@send char <ASCII character number>
@send ctrl '^<char>'
@send line <line to send (carriage return will be added)>
@send string <string of text>
```

'@send brk' will send the break signal (see chapter on sending the break signal)

'@send char' will send any character corresponding to the decimal character number provided; handy for sending characters which are non-printable.

'@send ctrl' will send a CTRL+<character> sequence

'@send line' will send a line of text, to which the a newline will automatically be added

'@send string' will send a string of text; newlines can be included in the text by specifying *"\n"*

Examples of where these can come in useful are for example when entering the debug shell of a device (which will not present a normal CLI prompt) and navigating within such interface. Also if one wanted to reboot the switch from the ACLI script, it would be safer to do:

```
@send line "reset -y"
```

As there might not be a CLI prompt coming back; then again, the connection will be lost if the switch is rebooted and the script will halt anyway (expect enhancements here to let the script carry on after the reboot...)

Launching a script with socket tied terminals

ACLI scripting in conjunction with ACLI sockets is possible. The most common use would be to execute a script on the socket tied terminal and have the same script be executed on all the socket listening terminals. This is easily done, either with copy-pasting the script into the tied session or by using *@run* or *@source*. There is however a significant difference between the two approaches.

If we take the same CLI script example for creating our VLAN 666, and we paste that into the socket tied terminal, what will happen is that each and every command will get executed on the tied terminal and will also be sent to the listening terminals. So far so good. However, while the tied terminal is operating in scripting mode, the listening ones are not, because they are simply being fed one command at a time. So if the script being pasted into the tied terminal contains embedded commands which only operate in scripting mode (such as the control structures: *@if, @else, @while, @for, @loop, etc..*) then these will not get processed on the listening terminals; in other words this won't work properly. It will mostly work properly if your script is just a simple list of switch CLI commands. However, even in this case, if an error is encountered on one of the CLI commands, either in the tied terminal, or in any of the listening terminals, then the script execution will halt in the tied terminal and hence will peter out across all listening terminals as well.

The safe way of executing the same ACLI script from a socket tied terminal and have it executed across all listening terminals is to have that script as a run file or a script file and to execute it using the either the *@run* or *@source* embedded commands; or else to associate it to an alias as a semi-colon fragmented command and execute that alias. In this case what happens is that only the *@run* or *@source* or alias command gets sent to the listening terminals, which then effectively run the script independently. Now, if an error occurs in one of the terminals, script execution will only be halted in that terminal (if the socket echo mode is set to error and the error occurs in a listening terminal, then the tied terminal will also get to see the error and will halt). The tied terminal will also run the script, but once the script was executed in this manner, then the tied terminal will stop sending commands to the listening terminals, until the script execution has completed.

In some scripting applications it can be useful if the script can be executed on one terminal instance only, and then that script initiates "@socket tie" and "@socket echo all" and then can decide exactly which commands to execute locally + over the socket. For this purpose a '-o' switch can be appended to any CLI or embedded command and will ensure it gets sent over the socket in scripting mode. The same switch will ensure that the command is sent with socket echo mode "all", even if the globally set echo mode is "none" or "error". The -o switch can only be applied on the command before any redirection (to file or variable) and before any repeat option; any variables in the command will also be dereferenced before the command is sent to the socket. The -o switch can also be immediately followed by a number which will add a delay time to wait for output from the sockets: -o[N]

If instead the intention is for the script to send CLI commands only to listening sockets (without executing them locally), this can be done using the *'@socket send'* embedded command (refer to the socket section). It is also possible to spawn new connections from a script using the *@launch* embedded command, which will create new ACLI instances which can be made to listen on specific socket names. I.e. from a script it is possible to spin up a new connection with *@launch* and then drive it with *@socket send*; this is all a bit experimental though (i.e. might be buggy!).

Terminal Server caching

The ACLI terminal has a caching functionality for remembering past connections to terminal servers. A terminal server is a device one connects to via Telnet (or SSH) in order to get access to a switch serial console port. These connections are generally made by opening a Telnet (or SSH) connection against a non-default TCP port number. The terminal server will allocate TCP port numbers against each serial port connection it can connect to. This is all very nice, but the pain with connecting via terminal servers is that one never remembers the TCP port number to use and what switch console port is connected to which terminal server serial port.

Terminal Server connection caching

ACLI solves this problem by maintaining a cache of all past connections to terminal servers. The cache information is augmented with details of the connected device, such as switch name, switch model, switch MAC address, CPU information or stack/unit information. The terminal server cache information is stored in file *%USERPROFILE%\.acli\acli.trmsrv* and can be verified using the *'trmsrv info'* command under the ACLI control interface:

```
ACLI> trmsrv info
Remote terminal-server File : C:\Users\lstevens\.acli\acli.trmsrv
Sort mode               : ip
Static mode             : enabled
ACLI>
```

A new entry is added to the cache file whenever a new connection is made, via a terminal server, and that ACLI auto-discovery is run, resulting in ACLI entering interactive mode. The ACLI auto-discovery process allows ACLI to discover all the necessary switch information which is then added to the cache file. Note that it is usually a good idea to launch ACLI connections to a terminal server by setting the *-n* ACLI switch, to disable ACLI's auto-discovery upon connection (this is because ACLI's auto-discovery can be a bit slow, if the the connection is ultimately over a 9600 baud serial port connection, and may not work if the switch requires some key sequences before presenting a CLI prompt). In this case, the user will have to hit CTRL-T to force auto-discovery in order to get the terminal server connection to get cached.

The information in the cache file can be viewed using the *'trmsrv list'* command in the ACLI control interface:

```
ACLI> trmsrv list

Known remote terminal-server sessions:

Num  TrmSrv/IP ssh/tel   Port Name of attached device (details)
---  ---------------- ----  ---- ------------------------------------------------------------
  1  10.8.3.239         t  2029 VSP-4850GTS (VSP-4850-GTS; c0-57-bc-b2-f0-00; CPU1)
  2  192.168.0.200      t  5011 ERS4800-Stk (ERS-4826-GTS; 50-61-84-FB-D8-01; Unit2)
  3  10.8.10.239        t  5007 SrvDist1 (ERS-8806; 00-0f-06-c9-d0-00; CPU5)
  4  10.8.10.239        t  5012 VSP7000XT-6 (VSP-7024-XT; A0-12-90-03-E8-00; Standalone)
  5  10.8.5.239         t  7001 X670G2-48x-4q (X670G2-48x-4q; 00-04-96-A0-9B-D2)
  6  10.8.5.239         t  7014 ERS4800-1 (ERS-4826-GTS-PWR+; 50-61-84-FB-BC-00; Standalone)
  7  10.8.5.239         t  7007 ERS5900-STK (ERS-5928-GTS-uPWR; D4-78-56-07-FC-01; Unit1)
  8  10.8.5.239         t  7005 ERS5900-FC (ERS-5928-MTS-uPWR; 70-7C-69-05-84-00; Standalone
  9  10.8.5.239         t  7008 ERS4900-FC (ERS-4950-GTS-PWR+; B4-2D-56-53-CC-00; Standalone
 10  10.8.5.239         t  7013 ERS3600-STK (ERS-3626-GTS-PWR+; C4-BE-D4-72-51-01; Unit2)
 11  10.8.5.239         t  7016 ERS4800-STK (ERS-4826-GTS-PWR+; 50-61-84-FB-D0-01; Unit1)
 12  10.8.5.239         t  7011 ERS4900-STK (ERS-4950-GTS-PWR+; B4-2D-56-55-64-01; Unit1)
```

Not all connections made with Telnet/SSH to non-default TCP numbers are connections via a terminal server. For example, a real SSH connection could be made towards a switch, but have to pass via some intermediate device which is performing port-forwarding. ACLI will only cache connections which are deemed genuine terminal server connections. ACLI will consider a connection as a terminal server connection under the following cases:

- The connection protocol is either Telnet or SSH
- And, a non default TCP port number is used
- And, either the *-t* ACLI command line switch was set, indicating a terminal server connection
- or the *-n* ACLI command line switch was set, indicating a desire not to run auto-discovery upon connection

A connection is also automatically classified as a terminal server connection is it was set up with these syntaxes:

- *C:\> acli -n trmsrv:*
- *ACLI> open -n trmsrv:*
- *ACLI> trmsrv connect <index-number>*

These syntaxes will be covered below.

Connecting to a cached Terminal Server entry

So ACLI keeps a handy cache list of past terminal server connections. But how to leverage this list to make it easy to re-establish one of these past connections ? There are two possible approaches.

The first approach is to lauch ACLI using the following syntax:

```
C:\> acli -n  trmsrv:[<device-name>|<host/IP>#<port>] [<capture-file>]
```

The same syntax is also available via the *open* command under ACLI control interface:

```
ACLI> open -n  trmsrv:[<device-name>|<host/IP>#<port>] [<capture-file>]
```

Note that the *-t* switch does not need to be set with these commands, as a connection to a terminal server is exlpicit. Using the *-n* switch is optional, but highly recommended for the initial connection onto a switch console serial port.

The *'trmsrv:'* keyword can be followed by an optional string. The string is used to try and identify a valid entry in the terminal server cache file. For example, if the *<device-name>* string matches a single entry in the cache file, then that entry will be used to open a connection to the corresponding terminal server IP and TCP port. If instead the string is not provided or matches multiple entries in the cache file, then the following interactive menu will be displayed:

```
ACLI> open -n trmsrv:ers

Multiple entries match selection "ers"
Known remote terminal-server sessions matching 'ers':

Num  TrmSrv/IP ssh/tel  Port Name of attached device (details)
---  ---------------- ----  ------------------------------------------------------------
  2  192.168.0.200    t   5011 ERS4800-Stk (ERS-4826-GTS; 50-61-84-FB-D8-01; Unit2)
  3  10.8.10.239      t   5007 SrvDist1 (ERS-8806; 00-0f-06-c9-d0-00; CPU5)
  6  10.8.5.239       t   7014 ERS4800-1 (ERS-4826-GTS-PWR+; 50-61-84-FB-BC-00; Standalone)
  7  10.8.5.239       t   7007 ERS5900-STK (ERS-5928-GTS-uPWR; D4-78-56-07-FC-01; Unit1)
  8  10.8.5.239       t   7005 ERS5900-FC (ERS-5928-MTS-uPWR; 70-7C-69-05-84-00; Standalone
  9  10.8.5.239       t   7008 ERS4900-FC (ERS-4950-GTS-PWR+; B4-2D-56-53-CC-00; Standalone
 10  10.8.5.239       t   7013 ERS3600-STK (ERS-3626-GTS-PWR+; C4-BE-D4-72-51-01; Unit2)
 11  10.8.5.239       t   7016 ERS4800-STK (ERS-4826-GTS-PWR+; 50-61-84-FB-D0-01; Unit1)
 12  10.8.5.239       t   7011 ERS4900-STK (ERS-4950-GTS-PWR+; B4-2D-56-55-64-01; Unit1)
 18  192.168.0.199    t   5001 2550T-PWR (ERS-2550-T-PWR; 5C-E2-86-E8-40-00; Standalone)
 19  192.168.0.200    t   5005 ERS3600 (ERS-3626-GTS; C4-BE-D4-72-82-00; Standalone)
 23  192.168.0.200    t   5009 ERS5900 (ERS-5928-GTS; 00-1B-4F-FC-68-00; Standalone)
 24  192.168.0.200    t   5003 ERS4500 (ERS-4548-GT-PWR; 00-1C-9C-3F-88-00; Standalone)
 25  192.168.0.200    t   5004 ERS3500 (ERS-3549-GTS; B0-AD-AA-51-5C-00; Standalone)
 26  192.168.0.200    t   5002 ERS5600 (ERS-5650-TD-PWR; 5C-E2-86-28-E4-00; Standalone)
 27  192.168.0.200    t   5008 ERS4900 (ERS-4926-GTS-PWR+; 04-8A-15-60-88-00; Standalone)
 31  10.193.0.12      t   5015 Access-1 (ERS-5928-GTS-PWR+; 00-1B-4F-FC-F0-00; Standalone)

Select entry number / device name glob / # :
```

If no string was provided, all entries in the cache file will be listed. In the example above a string of *'ers'* was provided which matches multiple entries and so only those entries are listed. Note that the string is used to match the entries against any field, not just the switch name column.

At this point the user can either select one of the numbered entries, to connect to it, or can yet again enter a different search string to either narrow down the search even further or to look for some completely different entries. In the following example, a string of *'ers3'* is entered, resulting in:

```
Select entry number / device name glob / # : ers3

Known remote terminal-server sessions matching 'ers3':

Num  TrmSrv/IP ssh/tel  Port Name of attached device (details)
```

```
    --- ---------------- ---- ------------------------------------------------------------
     10  10.8.5.239       t   7013 ERS3600-STK (ERS-3626-GTS-PWR+; C4-BE-D4-72-51-01; Unit2)
     19  192.168.0.200    t   5005 ERS3600 (ERS-3626-GTS; C4-BE-D4-72-82-00; Standalone)
     25  192.168.0.200    t   5004 ERS3500 (ERS-3549-GTS; B0-AD-AA-51-5C-00; Standalone)

    Select entry number / device name glob / # :
```

Note, to come back to the full list, it is sufficient to provide a string consisting of just *'.'* (in Perl Regular expressions, the dot character matches any character)

We shall assume that we have found the connection we want, and we shall pick the entry 10, for the ERS3600-STK:

```
    Select entry number / device name glob / # : 10
    Logging to file: C:\Users\lstevens\Documents\ACLI-logs\10.8.5.239-7013.log
    Escape character is '^]'.
    Trying 10.8.5.239 port 7013 ...........
```

Note that ACLI will then connect to the corresponding IP and TCP port number.

Coming back to the original command, we could have equally launched the connection using this syntax:

```
    C:\> acli -n  trmsrv:ERS3600-STK
```

Or via the *open* command under ACLI control interface:

```
    ACLI> open -n  trmsrv:ERS3600-STK
```

In this case, the connection would be made immediately, since the string 'ERS3600-STK' matches only one entry in the cache file:

```
    ACLI> open -n trmsrv:ERS3600-STK

    Logging to file: C:\Users\lstevens\Documents\ACLI-logs\10.8.5.239-7013.log
    Escape character is '^]'.
    Trying 10.8.5.239 port 7013 ........
```

The string provided can also take the format *<entry|IP>#<port>*. This syntax is a bit more exotic.

- If string is in format *<entry>#<port>*, the corresponding entry number is looked up in the cache file, and the terminal server IP is extracted from this entry, but the TCP port to use will be the number provided after the '#' character. As a further twist, if the number provided is between 1 - 16, the TCP port used will be 5001 - 5016 (this is because the ACLI author still uses ancient Remote Annex terminal servers!)
- If string is in format *<IP>#<port>*, a connection is made to that IP address with a TCP port specifed after the '#' character. Again, if the number provided is between 1 - 16, the TCP port used will be 5001 - 5016.

The latter of the above syntaxes, is really no different from doing:

```
    ACLI> open -nt  <IP> <tcp-port>
```

The advantage of these syntaxes is that they can also be used at the *'Select entry number / device name glob / <entry|IP>#<port> :'* prompt. This is handy as one might want to use the cached list to find the IP address of the desired terminal server, but then specify a new TCP port to use with it.

The second appoach for using the terminal server cache list, is to simply use the *'trmsrv list'* and *'trmsrv connect'* commands under the ACLI control interface.

```
    ACLI> trmsrv list ?
    Syntax: trmsrv list [<pattern>]
    ACLI> trmsrv list ers

    Known remote terminal-server sessions matching 'ers':

    Num  TrmSrv/IP ssh/tel  Port Name of attached device (details)
```

```
--- ----------------  ---- ---------------------------------------------------------
  2  192.168.0.200     t   5011 ERS4800-Stk (ERS-4826-GTS; 50-61-84-FB-D8-01; Unit2)
  3  10.8.10.239       t   5007 SrvDist1 (ERS-8806; 00-0f-06-c9-d0-00; CPU5)
  6  10.8.5.239        t   7014 ERS4800-1 (ERS-4826-GTS-PWR+; 50-61-84-FB-BC-00; Standalone)
  7  10.8.5.239        t   7007 ERS5900-STK (ERS-5928-GTS-uPWR; D4-78-56-07-FC-01; Unit1)
  8  10.8.5.239        t   7005 ERS5900-FC (ERS-5928-MTS-uPWR; 70-7C-69-05-84-00; Standalone
  9  10.8.5.239        t   7008 ERS4900-FC (ERS-4950-GTS-PWR+; B4-2D-56-53-CC-00; Standalone
 10  10.8.5.239        t   7013 ERS3600-STK (ERS-3626-GTS-PWR+; C4-BE-D4-72-51-01; Unit2)
 11  10.8.5.239        t   7016 ERS4800-STK (ERS-4826-GTS-PWR+; 50-61-84-FB-D0-01; Unit1)
 12  10.8.5.239        t   7011 ERS4900-STK (ERS-4950-GTS-PWR+; B4-2D-56-55-64-01; Unit1)
 18  192.168.0.199     t   5001 2550T-PWR (ERS-2550-T-PWR; 5C-E2-86-E8-40-00; Standalone)
 19  192.168.0.200     t   5005 ERS3600 (ERS-3626-GTS; C4-BE-D4-72-82-00; Standalone)
 23  192.168.0.200     t   5009 ERS5900 (ERS-5928-GTS; 00-1B-4F-FC-68-00; Standalone)
 24  192.168.0.200     t   5003 ERS4500 (ERS-4548-GT-PWR; 00-1C-9C-3F-88-00; Standalone)
 25  192.168.0.200     t   5004 ERS3500 (ERS-3549-GTS; B0-AD-AA-51-5C-00; Standalone)
 26  192.168.0.200     t   5002 ERS5600 (ERS-5650-TD-PWR; 5C-E2-86-28-E4-00; Standalone)
 27  192.168.0.200     t   5008 ERS4900 (ERS-4926-GTS-PWR+; 04-8A-15-60-88-00; Standalone)
 31  10.193.0.12       t   5015 Access-1 (ERS-5928-GTS-PWR+; 00-1B-4F-FC-F0-00; Standalone)

ACLI> trmsrv connect ?
Syntax: trmsrv connect <entry-index-number>
ACLI> trmsrv connect 10

Logging to file: C:\Users\lstevens\Documents\ACLI-logs\10.8.5.239-7013.log
Escape character is '^]'.
Trying 10.8.5.239 port 7013 ............
```

The *'trmsrv list'* command also accepts a pattern string to only display matching entries. But the *'trmsrv connect'* command will only work by providing a discrete entry number from the cached entries file and will always make the connection without auto-detection.

Manually modifying the terminal server cache file

Entries in the cache file can be added using the following ACLI control interface command:

```
ACLI> trmsrv add telnet|ssh <IP/hostname> <TCP-port> <Device-Name> [<Comments>]
```

The command additionally allows a comment field to be added to the entries.

To delete an entry use the following ACLI control interface command:

```
ACLI> trmsrv remove telnet|ssh <IP/hostname> <TCP-port>
```

Sorting entries in the terminal server cache file

By default, entries in the terminal server cache file are displayed in the same order in which they were added to the file. Furthermore, if an exiting entry is updated, then the original entry is deleted and the new updated entry is appended to the file. This means that the last entries in the file will be the most recently added or updated entries.

It is however possible to sort the cache files, by setting a sort criteria via the *'trmsrv sort'* command:

```
ACLI> trmsrv sort cmnt|disable|ip|name
```

There are three sort criteria allowed:

- **ip** : Entries are sorted by terminal server IP address/hostname, then by TCP port
- **name**: Entries are sorted by switch name alone
- **cmnt**: Entries are sorted by comment field only

The sort setting is actually stored within the cache file (a line beginnig with *':sort'*). When the sorting method is set, the *':sort'* line is added to the cache file and all the entries in the file are re-arranged according to the sort criteria. If sorting is disabled, then the *':sort'* line is removed from the cache file.

Using a pre-filled cache file

It is possible to provide a pre-defined terminal server cache file, by simply editing the approproate *acli.trmsrv* text file. In this case it is preferable to place the edited, master copy, of the *acli.trmsrv* file in the ACLI install directory, where it will never be overwritten. When ACLI looks for the *acli.trmsrv* file it will search for it in the following paths in order:

- ENV path *%ACLI%* (if you defined it)
- ENV path *$HOME/.acli* (on Unix systems)
- ENV path *%USERPROFILE%\.acli* (on Windows)
- ENV path *%ACLIDIR%* (the ACLI install directory)

Whereas when ACLI updates or adds a new entry to the file, the file will be saved only to the first of the above paths (i.e. never in the last one, the ACLI instal directory).

So once a custom *acli.trmsrv* file has been placed in the ACLI install directory, it is necessary to delete any previously saved cache files in the user directories, otherwise the custom cache file will never get loaded. Deleting the previously cached versions of the file can be done with the following command:

```
ACLI> trmsrv delete file
```

This command will never delete the *acli.trmsrv* file in the ACLI install directory, but will delete any other *acli.trmsrv* files found in the other paths.

Once the custom *acli.trmsrv* cache file gets loaded from the ACLI install directory, any new or updated entries added to the file, will result in a new *acli.trmsrv* file being saved in the user's personal path (or the *%ACLI%*, if defined). This new file will keep being used and updated by ACLI now, but it will now contain all the entries from the original *acli.trmsrv* file which remains located in the ACLI install directory, and acts as a master copy.

To avoid having to go and delete the personal cached file every time the master copy file is updated it is possible to set the *master_trmsrv_file_str* key in *acli.ini*. This key should either point to a filename under the ACLI install directory or to the full path to any other file (if not located in the ACLI directory). Once set and the file exists, the date of the file will be compared with the date of the user's personal terminal-server file (under *%USERPROFILE%\.acli\acli.trmsrv*), if it exists, and whichever is the most recent will automatically be used.

Setting the static flag

The ACLI behaviour, when updating the terminal server cache file, is that any updated entry, which is found to have the same MAC addres as a previously cached entry (or it has the same switch name than an existing entry for which the MAC address was not recorded), will result in the existing entry being deleted from the cache file. This behaviuor of deleting existing entries, is not desireable if the entries were derived from a custom *acli.trmsrv* file. To prevent ACLI from deleting existing entries in the cache file it is possible to set a static flag for the cache file, using the following command:

```
ACLI> trmsrv static disable|enable
```

Much like the *'trmsrv sort'* setting, the static flag setting is actually stored within the cache file (a line beginnig with *':static'*). When the static flag is enabled, the *':static'* line is added to the cache file. If static flag is disabled, then the *':static'* line is removed from the cache file.

Acli.ini file

The ACLI terminal comes with a set of default settings and behaviours which can be overridden on startup using command line switches. However it can be annoying to remember every time to include the relative command line switches, also it becomes impractical to have too many command line switches.

For this reason ACLI comes with an *acli.ini* file where a number of keys can be set to permanently override ACLI's default behaviours. The ACLI distribution already includes an *acli.ini* file which can be found in the install directory. This file can be used as a template to create a custom ini file. In it are listed all the available keys and the syntax to use as well as a description of each key. The keys themselves are however all commented out and show the default value for the setting.

To create a custom *acli.ini* file two options are possible:

- The supplied *acli.ini* file in the ACLI install directory is simply edited and modified. This is not the preferred method, as this file is versioned and when running the ACLI update script, if a newer version of this file exists (e.g. if new ini keys are defined and need adding to the template) on the update server, the local copy of this file will be over-written (a temporary backup can be found in the updates/rollback directory). To prevent this from happening, one could edit the version number of the *acli.ini* file to 999 to ensure it never gets over-written by the update script. The next method is however preferred.
- Edit the supplied *acli.ini* file but save the modified file into one of the following directories:
    - ENV path *%ACLI%* (if you defined it)
    - ENV path *$HOME/.acli* (on Unix systems)
    - ENV path *%USERPROFILE%\.acli* (on Windows)

Follows a list of all the keys which can be set in the *acli.ini* file.

- **timeout_val**: Default timeout value in secs; used by underlying Perl module *Control::CLI*. Default is 10 seconds.
- **connect_timeout_val**: SSH & Telnet connection timeout - for establishing TCP connection. Default is 25 seconds.
- **login_timeout_val**: Timeout in seconds to apply to initial login (navigating through login banners etc, to obtain first CLI prompt). Default is 30 seconds.
- **peercp_timeout_val**: Default timeout value in secs for establishing peer CPU connection. Default is 4 seconds.
- **auto_detect_flg**: Flag which determines whether the ACLI terminal tries to auto detect the connected host type during connection and...
- **family_type_interact_flg:<family-type>**: ... then goes into interact mode, if the family type interact flag is set. By default family types which are supported by ACLI are set. The family type of the connected device can be viewed by inspecting the *$_family_type* attribute variable:

```
VSP-8284XSQ:1#% @vars attribute family

$_family_type           = PassportERS

VSP-8284XSQ:1#%
```

- **prompt_suffix_flg**: Flag which if set and if the ACLI terminal is in interactive mode, then the **prompt_suffix_str** will be appended to the switch prompt to indicate that the terminal has locked on the device prompt. Default is enabled.
- **prompt_suffix_str**: String which, if **prompt_suffix_flg** is enabled, is appended to connected device's CLI prompt to indicate that the ACLI terminal has locked on the CLI prompt. Default string is '% '
- **more_paging_flg**: Flag which if enabled, the ACLI terminal will by default have local more paging enabled upon connection. Default is enabled. (This is the same as performing *'@more enable'* once connected)
- **more_paging_lines_val**: Default number of lines displayed by ACLI's local more paging in each page of output. Default is 22.
- **alias_enable_flg**: Flag which enables command aliasing by default. Default is enabled.

- **alias_echo_flg**: Flag which, if command aliasing is enabled, determines whether an extra echo line is added to output to indicate how an alias was converted into a switch command. Default is enabled.
- **vars_echo_flg**: Flag which determines whether an extra echo line is added to output to indicate how variables were dereferenced into switch command. Default is enabled.
- **history_echo_flg**: Flag which determines whether an extra echo line is added to output to indicate which history recalled command (using *!*<*n*>) is being sent as switch command. Default is enabled.
- **dictionary_echo_flg**: Flag which determines whether or how an extra echo line is added to output to indicate how a dictionary command was translated. Default is 2 (single command)
- **ctrl_escape_chr**: Default CTRL character to break into ACLI control interface. Default is *'^]'*.
- **ctrl_quit_chr**: Default CTRL character to quit the ACLI terminal. Default is *'^Q'*.
- **ctrl_interact_toggle_chr**: Default CTRL character to toggle between ACLI interact and transparent modes. Default is *'^T'*.
- **ctrl_more_toggle_chr**: Default CTRL character to toggle between local more paging enabled and disabled. Default is *'^P'*.
- **ctrl_break_chr**: Default CTRL character to send the break signal to connected device. Default is *'^S'*.
- **ctrl_clear_screen**: Default CTRL character to clear the screen. Default is *'^L'*.
- **ctrl_debug_chr**: Default CTRL character to dump debug information (will only work if ACLI terminal is already in debug mode). Default is *'^['*.
- **grep_indent_val**: Number of SPACE characters to use when indenting an ACLI config (applicable to family types PassportERS & BaystackERS, where the ACLI terminal takes an active role in reformatting the output of "show running-config" with the -i switch set. Default is 3 spaces.
- **syntax_acli_mode_flg**: Flag which determines whether '?' behaves like in acli/nncli (1), in that the syntax of whatever partially entered command is automatically displayed (without having to hit the Enter key) or like with old Passport CLI (0) where the '?' character is treated

no differently from any other character (and user has to hit the Enter key to get command syntax). Default is enabled.

- **keepalive_timer_val**: Timer in minutes for sending keep alive carriage returns to prevent device from timing out session (0 to disable keepalives). Default is 4 minutes.
- **transparent_keepalive_flg**: Determines whether keep alive carriage returns should also be sent in transparent mode; 0 = no, only in interactive mode; 1 = yes, in both interactive and transparent modes; note that there is no suppression of prompts generated by keepalive in transparent mode. Default is 0
- **session_timeout_val**: Timer in minutes for holding up session to device (0 to disable session timeout). Default is 600 minutes (10 hours).
- **socket_enable_flg**: Enable socket functionality to drive many terminals from one. Default is enabled.
- **socket_bind_ip_str**: Local IP interface where to bind sockets (used for tie-ing terminals together). To bind to all available IP interfaces set to the empty string ''. Default is the loopback address '127.0.0.1'.
- **socket_send_ip_str**: Destination socket IP address used to send to, on controlling (tied) terminal. Default is the loopback broadcast address '127.255.255.255'.
- **socket_send_ttl_val**: IP TTL to use on socket IP Multicast packets; on loopback interface can stay at 0. Default is 0.
- **socket_send_username_flg**: Determines whether the username is encoded in socket datagrams; needed if socket functionality is used by different users on same machine. 0 = disabled; 1 = enabled. Default is enabled (1).
- **socket_allowed_source_ip_lst**: Comma separated list of socket source IP addresses from which listening terminals will accept commands. Default is list of one single IP address, the loopback IP: ['127.0.0.1'].
- **socket_names_val:all**: Socket port number base. The 'all' port is always defined and determines the starting UDP port number used for sockets. Note that if an *acli.sockets* file already exists, then the port numbers will be read from that file. Default is 50000.
- **socket_echo_mode_val**: Default local echo mode used in socket tied terminal: 0 = no echo from listening terminals; 1 = only errors

messages from listening terminals; 2 = all output from listening terminals. Default is 1.
- **pseudo_prompt_str**: Pseudo terminal prompt string. Default is 'PSEUDO#'.
- **source_error_detect_flg**: Flag which determines whether or not we want to pause sourcing of commands (in scripting mode) if an error is detected from the connected device. Default is enabled. (This is the same as using '@error enable')
- **source_error_level_str**: Error detection (in scripting mode) can be set either for 'error' or just 'warning'. An error condition is when the CLI command sent to the connected device was not executed and an error message was produced by the host. A warning condition is when the CLI command sent to the connected host was executed but still generated a warning message. Default is 'error'. (Note that 'warning' mode is not fully implemented).
- **newline_chr**: Newline character to send to connected host; can be set to "\n" (CR+LF) or "\r"(CR). Default is "\r".
- **terminal_emulation_str**: Default terminal emulation type negotiated during underlying Telnet or SSH connection. Default is 'vt100'. (Does not really matter for most Extreme devices).
- **terminal_window_size_lst2**: Default terminal window size (terminal width, lines per page) negotiated during Telnet or SSH connection. Note that this is not related to the window size of the ConsoleZ application window. It only matters with certain Extreme devices (to date, the WLAN9100 family type) which actually uses these negotiated values for command line scrolling & more paging. Default is [132, 24].
- **highlight_fg_colour_str**: Default highlight foreground text colour. Accepted values are: black|red|green|yellow|blue|magenta|cyan|white|disable. Default is 'red'.
- **highlight_bg_colour_str**: Default highlight background text colour. Accepted values are: black|red|green|yellow|blue|magenta|cyan|white|disable. Default is 'disable'.
- **highlight_text_bright_flg**: Flag which determines whether the highlighted text is to be rendered as bright. Default is enabled.

- **highlight_text_underline_flg**: Flag which determines whether the highlighted text is to be underlined. Default is disabled.
- **highlight_text_reverse_flg**: Flag which determines whether the highlighted text is to be rendered in reverse mode. Default is disabled.
- **ssh_default_keys_lst**: List of SSH Private keys to try and load on startup; the first key found will be loaded. Provide either the key basename(s) or full key path(s). Default is [*'id_rsa'*, *'id_dsa'*]. If just basenames are provided, the keys will be searched in these paths:
  - *%ACLI%\.ssh* (if you defined it)
  - *$HOME/.ssh* (on Unix systems)
  - *%USERPROFILE%\.ssh* (on Windows)
  - *%ACLIDIR%\.ssh* (the ACLI install directory)
- **quit_on_disconnect_flg**: Flag which determines behaviour when the connection is lost: 0 = offer to reconnect; 1 = quit. Default is 0.
- **working_directory_str**: Default working directory to use for all connections. Default is unset, with empty string ''.
- **auto_log_to_file_flg**: Automatically log session to file: 0 = disabled; 1 = enabled. Default is disabled (0).
- **log_path_str**: If set, determines the default path where session log files (whether using auto-log or not) will be created. If no log path is set, then files will be created in the current working directory. Default is unset, with empty string ''.
- **auto_log_filename_str**: If auto-log is enabled, the filename used will be the device IP or hostname (whichever was provided to make the connection). Via this setting it is possible to pre-pend or append a timestamp to the actual filename. If not set (set to empty string '') the auto-log filename will simply be the ip/hostname of the connection. The "<>" characters represent the ip/hostname portion of the filename, so the timestamp can be either pre-pended or appended as desired; if no "<>" characters are included then the timestamp will be pre-pended. To format the desired timestamp, provide a string supported by POSIX strftime (see http://perldoc.perl.org/POSIX.html#strftime). Default is '%Y_%m_%d-%Hh%Mm%Ss-<>'.
- **master_trmsrv_file_str**: Master terminal-server file. This key should either point to a filename under the ACLI install directory or to the full path to any other file (if not located in the ACLI directory). If set and a the file exists, the date of the file will be compared with the date of the

user's personal terminal-server file (*%USERPROFILE%\.acli\acli.trmsrv*), if it exists, and whichever is the most recent will be used. Setting this key allows any modification of the master terminal-server file to automatically result in it being used next time without having to execute *'trmsrv delete file'* under the ACLI control interface. Do not set this key to the acli.trmsrv file under your personal directory (*%USERPROFILE%\.acli*); not setting this key will result in that file always being used if present anyway.

- **hide_timestamps_flg**: Flag which determines whether ACLI should supporess device timestamp banners from the output displayed on screen (the timestamp banners will still get recorded in ACLI logging files), 0 = disable; 1 = enable. Default is 0.
- **port_ranges_span_slots_flg**: Port ranges to span slots. On some devices, like VOSS and XOS, the following port ranges are allowed: 1/1-2/48 (or 1:1-2:48). While undeniably more compact, ACLI will default not to show ranges spanning slots, and will only create ranges within the same slot, like 1/1-1/48,2/1-2/48, which is more easily readable and the last port of each slot is easily recognizeable. 0 = port ranges do not span slots; 1 = port ranges can span slots, on devices which support it. Default is 0.
- **default_port_range_mode_val**: On devices which do not support port ranges (e.g. SecureRouter or WLAN APs generally) this setting determines how ACLI will display port lists (stored in variables). Valid settings are: 0 = no range just comma separated list; 1 = ranges in format 1/1-24; 2 = ranges in format 1/1-1/24.
- **port_ranges_unconstrain_flg**: ACLI will always process port ranges based on actual ports discovered on the connected switch. A port range expanded into a list ($var) will only include ports which actually exist on the connected switch. There are some script uses where it is handy if ports can be captured from output received over tied sockets from other switches by disabling this parameter. If enabled port ranges will only handle slot/port (are not processed: 'ALL', slot/port/chann, insight 1/s ports). 0 = disabled; 1 = enabled. Default is disabled (0).
- **highlight_entered_command_flg**: Flag which determines whether user entered (or pasted) commands, in interactive mode, are to be highlighted by making them bright.

- **ssh_known_hosts_key_missing_val**: SSH known hosts missing key behaviour. This setting determines how ACLI should behave when connecting via SSH to a host for which the key is missing in the known_hosts file. Valid settings are: 0 = SSH connection is refused; 1 = User gets interactively prompted whether to add the key for the host in the known_hosts file, or to connect once without adding the key to known_hosts, or to abort the connection; 2 = The key is automatically added to known_hosts file and a message is displayed to this effect. Default is 1.
- **ssh_known_hosts_key_changed_val**: SSH known hosts failure check behaviour. This setting determines how ACLI should behave when connecting via SSH to a host for which the key is already present in the known_hosts file, but the key offered by the host does not match the key cached in the known_hosts file. Valid settings are: 0 = SSH connection is refused; 1 = User gets interactively prompted whether to update the key for the host in the known_hosts file, or to connect once without updating the key in known_hosts, or to abort the connection; 2 = The key is automatically updated with the new key in the known_hosts file and a message is displayed to this effect. Default is 1.

Saving device related ACLI settings

The ACLI terminal offers a number of useful features and, for some of these, it is desireable to make them persistent whenever connecting to the same device. For example, when redirecting output to the local file system, one will typically set a working directory to a specific path. It would be annoying if one had to go set the working directory every time one connected to a given device. Also, if you are setting up variables for the devices, you probably would like these variables to be set as you left them next time you connect to the same device. And if you are using the socket functionality to drive multiple ACLI terminals from one driving terminal, you have probably assigned certain listening socket names to the various devices; again it would be annoying to have to set all this up every time we connected to the same devices.

The ACLI terminal, in interactive mode, allows the user to save all the above settings on a per device basis. This is achieved using the *@save* embedded command which will create a file *%USERPROFILE%\.acli\.vars\<device-MAC>.vars* with the relevant settings.

```
VSP-8284XSQ:1#% @save ?
Syntax: @save all|delete|info|reload|sockets|vars|workdir
```

The *@save* command offers a number of options to save all settings or just the working-dir or just the sockets or just the variables. It is usually safer to perform a *'@save all'* which will save all settings (performing one of the other options will create a new vars file which will contain only the specified settings of that type, and thus lose the other settings). The following example illustrates how these various settings can be set for the first time on a deive and then *@save*-ed.

```
VSP-8284XSQ:1#% @save inf
No save file for connected device

VSP-8284XSQ:1#% @$
No variables are set

VSP-8284XSQ:1#% @socket info

Socket settings:
        Socket functionality    : enable
        IP for sockets          : 127.255.255.255
        Tied to socket          :
        Local Echo Mode         : error
        Listening to sockets    :
        Socket Name File        :
        Bind to IP interface    : 127.0.0.1
        Allowed source IPs      : 127.0.0.1

VSP-8284XSQ:1#%
VSP-8284XSQ:1#% ifup > $up
        alias% show interfaces gigabitEthernet interface ||up\s+up > $up
===========================================================================================
                                      Port Interface
===========================================================================================
PORT                            LINK  PORT            PHYSICAL            STATUS
NUM     INDEX DESCRIPTION       TRAP  LOCK    MTU     ADDRESS             ADMIN  OPERATE
-------------------------------------------------------------------------------------------
1/1     192   10GbNone          true  false   1950    00:51:00:ca:e0:00 up       up
1/2     193   10GbNone          true  false   1950    00:51:00:ca:e0:01 up       up
1/3     194   10GbNone          true  false   1950    00:51:00:ca:e0:02 up       up

Var $up = 1/1-1/3

VSP-8284XSQ:1#% listen vsp
        alias% @socket listen add vsp
Listening on sockets: vsp

VSP-8284XSQ:1#% @cd C:\Users\lstevens\Downloads
```

```
New working directory is:
C:\Users\lstevens\Downloads

VSP-8284XSQ:1#%
VSP-8284XSQ:1#% @$

$up          = 1/1-1/3

Unsaved variables exist


VSP-8284XSQ:1#% @socket info

Socket settings:
        Socket functionality    : enable
        IP for sockets          : 127.255.255.255
        Tied to socket          :
        Local Echo Mode         : error
        Listening to sockets    : vsp(50079)
        Socket Name File        : C:\Users\lstevens\.acli\acli.sockets
        Bind to IP interface    : 127.0.0.1
        Allowed source IPs      : 127.0.0.1

VSP-8284XSQ:1#% @pwd

Working directory is:
C:\Users\lstevens\Downloads

VSP-8284XSQ:1#%
VSP-8284XSQ:1#% @save all
Variables, open sockets & working directory saved to:
 C:\Users\lstevens\.acli\.vars\00-51-00-ca-e0-00.vars

VSP-8284XSQ:1#% @save info

C:\Users\lstevens\.acli\.vars\00-51-00-ca-e0-00.vars:

# acli.pl saved on Fri Aug 17 22:48:01 2018
# Device base MAC    : 00-51-00-ca-e0-00
# Device sysname     : VSP-8284XSQ
# Device ip/hostname : 192.168.56.71

:wd          = C:\Users\lstevens\Downloads
:sockets     = vsp
$up          = 1/1,1/2,1/3


VSP-8284XSQ:1#%
```

An alias *'svv'* is also predefined, which performs a save config on the device + a *'@save all'*

Next time we connect to the same device, all those settings are reloaded automatically (notice the *'Loading var file'* message during device detection):

```
EXTREME NETWORKS VOSS COMMAND LINE INTERFACE

Login: rwa
Password: ***

acli.pl: Detected an Extreme Networks device -> using terminal interactive mode
VSP-8284XSQ:1>% enable
acli.pl: Detecting device ...
acli.pl: Detected VSP-8284-XSQ (00-51-00-ca-e0-00) Single CPU system, 2 slots 84 ports
acli.pl: Loading var file C:\Users\lstevens\.acli\.vars\00-51-00-ca-e0-00.vars
acli.pl: Use '^T' to toggle between interactive & transparent modes

VSP-8284XSQ:1#% @$

$up          = 1/1-1/3
```

```
VSP-8284XSQ:1#% @socket info

Socket settings:
        Socket functionality    : enable
        IP for sockets          : 127.255.255.255
        Tied to socket          :
        Local Echo Mode         : error
        Listening to sockets    : vsp(50079)
        Socket Name File        : C:\Users\lstevens\.acli\acli.sockets
        Bind to IP interface    : 127.0.0.1
        Allowed source IPs      : 127.0.0.1

VSP-8284XSQ:1#% @pwd

Working directory is:
C:\Users\lstevens\Downloads

VSP-8284XSQ:1#%
```

Note that any settings forced when launching acli via command line/shell will take precedence over values saved with @*save* in the device vars files. So, for example, if ACLI is invoked with -*w* <*work-dir*> or -*s* <*sockets*> then any working directory or socket information will not get loaded from the @*save* created vars files.

Peer CP functions

The ACLI terminal has some embedded functionality to deal with Extreme chassis based systems where both a Master Control Plane (CP) and a Standby CP are present. This functionality was developped primarily for operating with the PassportERS 8600/8800 series, where doing software upgrades with HA-mode enabled was a bit tricky in that the Standby CP had to be reset independently or simultaneously to the Master CP. It will however also work on more recent VSP9000 and VSP8600 platforms. Essentially the peer CP functionality adds two command line switches which can be added to any CLI command executed against the switch: -peercpu & -bothcpus:

- **-peercpu**: Execute the entered CLI command only against the peer CP (not against the Master CP). The advantage of this switch is that it saves the user from having to perform a *'peer telnet'* onto the Standby CP to execute the command there, and then logout to come back to the Master CP session. Will also work with *-peer* shortform.
- **-bothcpus**: Execute the entered CLI command against both the Master CP and the Standby CP simultaneously (in fact the command is executed on the Standby CP slightly before executing it on the Master CP, as typically one wants to reboot the whole chassis, and if the Standby CP is not rebooted first, then we instead achieve an HA-mode switchover). Will also work with *-both* shortform.

Both the above command line switches are only available if connected in interactive mode on the Master CP of a chassis with dual CPs and operate by setting up, in the background, a separate and independent CLI connection to the Standby CP. The functionality is clever enough to figure out whether we are connected to the chassis via Out-of-Band (OOB) and if so, whether the Standby CP has a dedicated OOB IP to which the connection can go directly; if not, then a second connection to the Master CP is setup and from there, a *'peer telnet'* is used to land on the Standby CP. The peer CP connection is automatically setup at the first use of the -peercpu or -bothcpus switches and will remain in place indefinitely thereafter. Alternatively the peer CP connection can be managed using the embedded @peercp command or using the 'peercp' commands under the ACLI control interface.

Example using *-peer* to view the boot config on the Peer CPU:

```
ERS8800-2:5#% show bootconfig config -peer
Connecting to peer CPU ..........................
Output from Peer CPU:
#
# SUN OCT 07 10:46:11 2018 UTC
# box type : 8k boot configuration file
```

```
#
flags ftpd true
flags ha-cpu true
flags rlogind true
flags savetostandby true
flags sshd true
flags telnetd true
flags tftpd true
flags verify-config false
choice primary image-file "/flash/p80ae72260.img"
choice secondary image-file "/pcmcia/p80ae7101.img"
net mgmt ip 10.8.10.17/255.255.224.0 cpu-slot 6
net mgmt ip 10.8.10.16/255.255.224.0 cpu-slot 5
delay 10
mezz-image image-name "/pcmcia/p80m72190.img"
@ERS8800-2:6#
ERS8800-2:5#%
```

In the above example, the connection to the peer CP was not already in place, so the connection is setup there and then.

The connection to the peer CP can also be brought up manually (or disconnected) using the @*peercp* embedded command or the *peercp* command under the ACLI control interface:

```
VSP8600-1:1#% @peercp ?
Syntax: @peercp [connect|disconnect]

VSP8600-1:1#% @peercp connect
Connecting to peer CPU ...
Connected to Peer CPU via shadow connection to 10.8.2.2
```

Example using -*both* to view the timezone settings across both CPUs:

```
ERS8800-2:5#% show bootconfig tz -both
Output from Peer CPU:
tz dst-end M10.5.0/0200
tz dst-name "UTC"
tz dst-offset 60
tz dst-start M4.1.0/0200
tz name "UTC"
tz offset-from-utc 0
TIMEZONE=UTC:UTC:0:::0
@ERS8800-2:6#

Output from Master CPU:
tz dst-end M10.5.0/0200
tz dst-name "UTC"
tz dst-offset 60
tz dst-start M4.1.0/0200
tz name "UTC"
tz offset-from-utc 0
```

```
        TIMEZONE=UTC:UTC:0:::0
        ERS8800-2:5#%
```

In the above example, the connection to the peer cp was already in place, so the output from both CPs is immediately available.

Example to reboot both ERS8800 CPUs in an HA-mode enabled configuration:

```
        ERS8800-1:5#% reset -both
        Connecting to peer CPU ..................................
        Output from Peer CPU:

        Output from Master CPU:
        ERS8800-1:5#%
        Received eof from connection

        ------> Connection closed: SPACE to re-connect / Q to quit <------
```

The status of the peer cp connection can be viewed using either the *@peercp* embedded command or the *peercp* command under the ACLI control interface.

```
        ERS8800-1:5#% @peercp
        Directly connected to Peer CPU on OOB IP 10.8.9.17
```

In the above example, the peer CP connection was made directly to the Standby CPU OOB IP interface.

```
        VSP8600-1:1#% @peercp
        Connected to Peer CPU via shadow connection to 10.8.2.2
```

In the above example, the peer CP connection was made via a second connection to the Master CP and, from there, into the Standby CP (using *peer telnet*). This can be seen by inspecting the connected users:

```
        VSP8600-1:1#% show users
        SESSION    USER                ACCESS    IP ADDRESS
        Console                        none      ----------
        SSH0       rwa                 rwa       10.8.0.158 (current)
        SSH1       rwa                 rwa       10.8.0.158
        VSP8600-1:1#%
```

Terminal emulation

The ACLI terminal will by default negotiate for a vt100 terminal and a terminal size of 132x24 width/height. Though, unlike most other terminals, this has absolutely no correlation to how large you make the ConsoleZ window in which ACLI is running. Negotiation is performed over both SSH and Telnet and this is taken care of by the underlying *Control::CLI* module. The defaults can be overridden using the *-y <term-type>* & *-z <w>x<h>* switches when starting acli from the command line/shell. They can also be changed using the ACLI control interface using the *'terminal'* command, however these settings will only apply to new connections (not an existing connection). The vt100 & 132x24 defaults can also be overridden in the *acli.ini* file via the *terminal_emulation_str* and *terminal_window_size_lst2* keys; see the ACLI ini file section.

Most devices will not care much about the terminal emulation in use. Though some will care. In the case of supported Extreme Networks devices, only the WLAN9100 cares about the negotiated terminal screen size which is then taken into account by the AP's CLI. The goal of the ACLI terminal is to treat all device output as simple text, where if lines are too large to fit in the window, the lines are wrapped by ConsoleZ/ACLI and not by the host device itself (if this happens ACLI has to try and unwrap them, otherwise this would compromise the ACLI grep functions). If a device takes notice of negotiated screen width, then setting this to 132 helps prevent that device from wrapping lines > 80 characters.

Sending Break signal

The break signal is outside the ASCII character set but has local meaning on some end systems (for example on some Cisco systems it is necessary to stop the boot sequence). The intention is to indicate that the Break Key or the Attention Key (on older terminals) was hit. The break signal is well defined over serial port RS232 communication and Telnet. It is less well defined over SSH. The ACLI terminal relies on the break signal implementation offered by the underlying *Control::CLI* module, which implements the break signal over serial RS232 using a 300ms pulse_break and over Telnet using the appropriate telnet option. Over SSH, the *Control::CLI* class currently simply sends *'~B'* in the data stream, though it is not clear if any SSH implementation supports this.

There are multiple ways to send the break signal. You can use either the *'@send break'* embedded command or the *'send break'* command under the ACLI control interface.

```
VSP-8284XSQ:1#% @send brk
VSP-8284XSQ:1#%

ACLI> send brk
ACLI>
```

However, the break signal typically needs to be sent while the connected device is booting up, at a very precise moment, so issuing an ACLI embedded command will not be possible and entering into the ACLI control interface may take too much time. for this reason the ACLI terminal pre-allocates a CTRL key to generate the break signal. By default the break signal control sequence is CTRL-S; this can however be changed in the *acli.ini* file. See the ACLI ini file section.

```
ACLI> ctrl info
CTRL characters:
        Escape character     : ^]
        Quit character       : ^Q
        Terminal mode toggle : ^T
        More paging toggle   : ^P
```

```
            Send Break              : ^S
            Debug                   : ^[
ACLI>
```

Pseudo terminal mode

Pseudo terminal mode is a way to make the ACLI terminal run in interactive mode without any host device connection present. It was mainly a mode used by the ACLI author to debug ACLI bugs/problems offline (without having to be connected to a device). To enter pseudo mode use either of the following:

- Execute ACLI from shell/command line using syntax: *acli pseudo:[<name>]*:

```
C:\Users\lstevens\Scripts\acli\working-dir>acli pseudo:
Loading alias file: C:\Users\lstevens\Scripts\acli\acli.alias
Merging alias file: C:\Users\lstevens\.acli\merge.alias
PSEUDO#%
```

- From the ACLI control interface, execute the *'pseudo'* command:

```
ACLI> pseudo ?
Syntax: pseudo attribute|disable|echo|enable|info|list|load|name|port-range|prompt|ty
ACLI> pseudo enable
PSEUDO#%
```

Pseudo mode can be useful in emulating connected switches, either to use the grep functionality on multiple offline switch config file, or if using the dictionary functionality to translate config files in offline mode.

A pseudo terminal can be named (earlier implementation would allow assigning a number, from 1-99 and default was 100, but these become names now), assigned a prompt to replace the default *PSEUDO#* prompt, assigned a Family Type, assigned attributes and a valid port-range.

In the following example, a profile is created to emulate a 5520 switch with VIM module ports and in VOSS persona:

```
PSEUDO#% @pseudo type ?
Syntax: @pseudo type boss|slx|voss|xos

PSEUDO#% @pseudo type voss
PSEUDO#% @pseudo port-range 1/1-48,2/1-4

Port Range: 1/1-1/48,2/1-2/4

PSEUDO#% @pseudo prompt 5520-24W#
5520-24W#% @pseudo name 5520-48W-VIM-VOSS

Pseudo terminal name(/id) set. To save terminal use '@save all'

5520-24W#%
```

This has now setup the following pseudo settings and attributes:

```
5520-24W#% @pseudo info

        Pseudo Terminal      : enabled
        Pseudo Name/Id       : 5520-48W-VIM-VOSS
        Pseudo Prompt        : 5520-24W#
        Pseudo Command Echo  : disabled
        Pseudo Family Type   : PassportERS
        Pseudo ACLI/NNCLI    : Yes
        Pseudo Port Range    : 1/1-1/48,2/1-2/4

5520-24W#% @pseudo attribute info

{is_ha}         = 0
{ports}         = ARRAY(0x37a8694)
{is_acli}       = 1
```

```
{cpu_slot}        = 1
{family_type}     = PassportERS
{is_master_cpu}   = 1
{is_dual_cpu}     = 0
{is_voss}         = 1
{slots}           = ARRAY(0x37a867c)

5520-24W#% @vars attribute

$_is_ha               = 0
$_ports               = ,ARRAY(0x4610ca4),ARRAY(0x4610d4c)
$_is_acli             = 1
$_cpu_slot            = 1
$_family_type         = PassportERS
$_is_master_cpu       = 1
$_is_dual_cpu         = 0
$_is_voss             = 1
$_slots               = 1,2

5520-24W#%
```

Notice that the base attributes of a VOSS switch are automatically set, as they would be if a real switch was connected. The attributes are important so that alias and dictionary commands can be correctly dereferenced/translated as if the real switch was connected. Additional attributes can be manually set using the *@pseudo attribute set <name> = <value>* command.

We can now save this Pseudo terminal profile using the *@save* command:

```
5520-24W#% @save all
Variables, open sockets & working directory saved to:
 C:\Users\lstevens\.acli\.vars\pseudo.5520-48W-VIM-VOSS.vars

5520-24W#%
```

Which will save all the Pseudo profile settings and attributes to file, as well as the usual variables, working directory and listening sockets if applicable:

```
5520-24W#% @save info

C:\Users\lstevens\.acli\.vars\pseudo.5520-48W-VIM-VOSS.vars:

# acli-dev.pl saved on Sun Feb 14 21:32:25 2021
# Pseudo Terminal   : 5520-48W-VIM-VOSS
:prompt           = 5520-24W#
:cmdecho          = 0
:family-type      = PassportERS
:acli-type        = 1
:port-range       = 1/1-1/48,2/1-2/4
:wd               = C:\Users\lstevens\Scripts\acli\working-dir
{is_ha}           = 0
{is_acli}         = 1
{cpu_slot}        = 1
{family_type}     = PassportERS
{is_master_cpu}   = 1
{is_dual_cpu}     = 0
{is_voss}         = 1


5520-24W#%
```

Already saved Pseudo terminal profiles can be listed using the *@pseudo list* command:

```
5520-24W#% @pseudo list
Available Saved Pseudo Terminals:

    Name                Origin   Family Type  Port Range
    ------------------  -------  -----------  ----------
    5520-48W-VIM-EXOS   private  ExtremeXOS   1-52
    5520-48W-VIM-VOSS   private  PassportERS  1/1-1/48,2/1-2/4
```

```
      X460G2                    private   ExtremeXOS

5520-24W#%
```

And any Pseudo profile can be loaded using either the *@pseudo load <name>* command or directly when launching ACLI from the command line:

```
C:\>acli pseudo:5520-48W-VIM-EXOS
Full Args: -d 0 pseudo:5520-48W-VIM-EXOS
Loading sed file: C:\Users\lstevens\Scripts\acli\acli.sed
Loading alias file: C:\Users\lstevens\Scripts\acli\acli.alias
Merging alias file: C:\Users\lstevens\.acli\merge.alias
Loading var file C:\Users\lstevens\.acli\.vars\pseudo.5520-48W-VIM-EXOS.vars
5520-48W-EXOS#% @pseudo info

        Pseudo Terminal     : enabled
        Pseudo Name/Id      : 5520-48W-VIM-EXOS
        Pseudo Prompt       : 5520-48W-EXOS#
        Pseudo Command Echo : disabled
        Pseudo Family Type  : ExtremeXOS
        Pseudo ACLI/NNCLI   : No
        Pseudo Port Range   : 1-52

5520-48W-EXOS#%
```

ACLI Spawn File

ACLI ships with a default *'acli.spawn'* file. This file is used to determine how best to spawn new ACLI terminal instances across any OS, such as Windows, MacOS or any of the Linux distributions. The file is inspected by the following ACLI applications:

- ACLI embedded command @*launch*; this allows an existing ACLI instance to spawn a new instance.
- ACLI GUI Script (acligui); this tool allows launching multiple ACLI instances at once, from a file, batch file, or the GUI itself. See the AcliGui entry under Other tools section.
- XMC ACLI Script (xmcacli); this tool is capable of extracting all the devices discovered in XMC, and to launch multiple ACLI instances at any of those devices. See the XmcAcli entry under Other tools section.

The above ACLI applications will all search for an *'acli.spawn'* file in the following directories in this order:

1. ENV path %ACLI% (if you defined it)
2. ENV path $HOME/.acli (on Unix systems)
3. ENV path %USERPROFILE%\.acli (on Windows)
4. Same directory where acli.pl resides

Note that the file is versioned and the version located in the same directory where acli.pl resides can get updated by the ACLI update script. Hence if you wish to modify it you should make a new version in one of the other paths (or you give it a huge 999 version).

The *'acli.spawn'* file shipped with ACLI contains detailed commentes. Essentially it allows to build a command line executable to spawn a new ACLI instance on a per OS basis, using a number of *<Tags>* which all of the above ACLI applications will replace with corresponding values at launch time. These tags are supported:

- **<WINDOW-NAME> :** [Optional] Name to assign to the new window
- **<INSTANCE-NAME> :** [Optional] If tabs are supported within the window, this is used as an identifer to the containing window where to open subsequent tabs
- **<TAB-NAME> :** [Optional] Name to assign to a tab within the window
- **<CWD> :** [Optional] Working directory to assign
- **<ACLI-PROFILE> :** [Mandatory on MSWin32] Tab profile to launch for ACLI (only applicable to Console.exe)
- **<ACLI-PATH> :** [Optional] Path to ACLI executable (batch or shell file, without .pl extension)
- **<ACLI-PL-PATH> :** [Optional] Path to acli.pl script file
- **<ACLI-ARGS> :** [Mandatory] Argument to pass to ACLI

Each line entry consists of 4 space separated values:

1. The OS version, as reported by Perl's $^O
2. An optional value in format N or F:N; where both N & F are decimal values (in range 0-9999) representing milliseconds to wait between every execution, where
   - F : Timer to wait between 1st and 2nd ACLI launch (required with Console.exe as it takes some time for the app to first launch)
   - N : Timer to wait between all subsequent ACLI lauches
3. Executable, including path
4. The rest of the line will be treated as arguments to supply to the above executable

As an example, these are the entries in the default *'acli.spawn'* file shipped with ACLI

```
MSWin32  600:100   %ACLIDIR%\Console.exe     -reuse -t <ACLI-PROFILE> -i "<INSTANCE-NAME>" -w "<
darwin             $ACLIDIR/ttab             -t "<TAB-NAME>" -d "<CWD>" <ACLI-PATH> <ACLI-ARGS>
linux              /usr/bin/gnome-terminal   --tab --title "<TAB-NAME>" --working-directory "<CW
```

It is recommended to avoid changing settings for MSWin32 and MACOS (darwin) as the ACLI distributions for these supply all the necessary executables

Where this file becomes really useful is when using the above ACLI applications on Linux systems, where there is a huge variety of desktop environments, each with a different executable to open up a terminal window.

For debugging use of the *acli.spawn* file, enable debug mode on ACLI, or set the *-d* command line switch on *acligui* or *xmcacli.*

Dictionary

The ACLI dictionary funtionality allows input commands to be accepted in the form of a different CLI syntax from the syntax of the connected switch. This works by loading a dictionary file where the input commands in one CLI lingo can be translated into other CLI flavours, of which the ACLI terminal will select the desitnation flavor based on what switch type it is connected to. Initially an ERS dictionary file is provided, which translates a selection of the most used BaystackERS commands into either PassportERS VOSS commands or ExtremeXOS commands. The ERS dictionary file can be easily extended to cover more commands and additional dictionary files can be added.

Dictionary functionality introduces a new *@dictionary* embedded command as well as a *'dictionary'* command under the ACLI control interface:

```
VSP-8284XSQ:1#% @dictionary ?
Syntax: @dictionary echo|info|list|load|path|port-range|reload|unload
```

Loading Dictionary Files

Dictionary files should always have a *.dict* extension and should be located under one of the following paths:

- ENV path *%ACLI%* (if you defined it)
- ENV path *$HOME/.acli* (on Unix systems)
- ENV path *%USERPROFILE%\.acli* (on Windows)
- Same directory where acli.pl resides (ENV path %ACLIDIR%)

The available paths under which dictionary files will be looked for can be viewed using the *@dictionary path* embedded command:

```
VSP-8284XSQ:1#% @dictionary path

Paths for Dictionary files:

   Origin    Path
   ------    ----
   private   C:\Users\lstevens\.acli
   package   C:\Users\lstevens\Scripts\acli

VSP-8284XSQ:1#%
```

Available dictionary files can be listed using the *@dictionary list* embedded command:

```
VSP-8284XSQ:1#% @dictionary list
Available Dictionaries:

   Name        Origin    Vers  Description
   ----        ------    ----  -----------
   ERS         package   0.03  BOSS/ERS Dictionary file

VSP-8284XSQ:1#%
```

Dictionary files are categorized as either:

- **Package**: These are dictionary files located under the ACLI install directory; typically these are dictionaries which are shipped with ACLI, and can be updated whenever the ACLI update script is executed, if a more recent version of the dictionary is available.
- **Private**: These are dictionary files located in the user's private path (either *%USERPROFILE%\.acli* or *$HOME/.acli* or *%ACLI%* if it was defined); this is where the user should place his or her dictionaries.

The *@dictionary list* command also extracts the dictionary version, if included (dictionary file should simply contain a line like *'# Version = <version>'*), and a description of the dictionary (this needs to be the first commented line inside the dictionary file). If you have many dictionary files, this becomes a neat way for keeping track of them all!

To load a dictionary file, simply use the *@dictionary load <name>* embedded command, where *<name>* is the name of the dictionary as displayed by the *@dictionary list* command (the *'.dict'* extension is implied and does not need to be specified). In this example the suplied ERS dictionary file is loaded:

```
VSP-8284XSQ:1#% @dictionary load ers
Loading dictionary file: C:\Users\lstevens\Scripts\acli\ers.dict

VSP-8284XSQ:1#{ERS}% @echo off output off
Sourcing ERS Dictionary script

Which VSP UNI config will apply for dictionary translations; CVLAN-UNI (1) or Switched-UN
VSP-8284XSQ:1#{ERS}%
```

Dictionary files can include an embedded script, which will get executed when the dictionary file is loaded. This will be detailed further down.

To be noticed, once a dictionary file is loaded, the ACLI intercative prompt will include the loaded dictionary name in curly brackets, here *{ERS}*

Information about loaded dicrionary can also be inspected via the *@dictionary info* embedded command:

```
VSP-8284XSQ:1#{ERS}% @dictionary info

Dictionary settings:
        Loaded dictionary : ERS
        Dictionary echoing: single
        Dictionary file   : C:\Users\lstevens\Scripts\acli\ers.dict
        Input Port Range  :
        Mapped Host Ports :

VSP-8284XSQ:1#{ERS}%
```

A reload option is also available:

```
VSP-8284XSQ:1#{ERS}% @dictionary reload
Loading dictionary file: C:\Users\lstevens\Scripts\acli\ERS.dict

Sourcing ERS Dictionary script

VSP-8284XSQ:1#{ERS}%
```

The reload option will re-read the dictionary file. This is handy if translations have been edited in the dictionary file and it is desired to simply update those definitions in the ACLI session. The dictionary script will also get re-executed, but the script itself can be written so as to decide what to do, or not to do, upon reload. Note that dictionary variables will not be cleared by a reload.

To completely unload a dictionary file, and clear out all dictionary variables use the *@dictionary unload* embedded command:

```
VSP-8284XSQ:1#{ERS}% @dictionary unload
VSP-8284XSQ:1#%
```

Dictionary File Structure

Dictionary files need to be edited with a specific structure and syntax. Much of the dictionary functionality was built re-using the ACLI alias functionality, so in many respects the syntax of dictionary files is similar to that of an alias file. Much of what documented in this section can also be found in the actual dictionary files shipped with ACLI.

Lines commencing with '#' are comment lines and are ignored. In dictionary files, comments can also be placed at the end of valid lines.

There are four parts to a dictionary file:

1. **Description:** This should be the very first commented line; this line will show when executing *@dictionary list*
2. **Dictionary version:** This should follow the first line providing a version for the dictionary file in format: *# Version = <version>*. The version number is shown when executing *@dictionary list* and, if the file was shipped with ACLI, used by the ACLI update script to determine if a newer version of this file exists
3. **Script section:** Used to declare dictionary varialble scope and to execute an arbitrary script upon loading the dictionary file. This section will include all non-commented lines before line **DICT_BEGIN**
4. **Dictionary section:** Section where every dictionary command is defined with its possible translations. This section will include all non-commented lines after line **DICT_BEGIN**

The script section can contain an ACLI script which will get executed when the dictionary file is first loaded and also when reloaded. The dictionary functionality sets aside a dedicated scope of ACLI variables which are needed to remember and store configuration information which cannot be immediately translated on the target lingo until subsequent input commands are entered. Also in some cases there are different ways to translate certain commands, depending on the desired target configuration. The dictionary script can thus both examine the connected device as well as take input from the user (when the dictionary file is loaded) and set any necessary dictionary variables to ensure proper translation of dictionary commands. Dictionary variables will be discussed in a section below.

The Dictionary section syntax will consits of a series of:

- Line with no spaces or tab prepended; this line is where a dictionary command is defined with all of its valid input syntax
- Then, on subsequent lines, a number of translation commands based on certain conditions. These translation lines must start with space or tab (i.e. indentation)

Additionally a special line *DICT_COMMENT_LINE = "<single-character>"* can be included to identify what character is used by the dictionary device to designate comment lines. As you might be pasting/sourcing config files from that device, we need the dictionary to know how to ignore comment lines.

If an invalid syntax is detected, ACLI will throw an error when trying to load the dictionary.

Dictionary command syntax:

- Enter dictionary command fully expanded and with all mandatory and optional sections listed in the right order. The defined syntax should cover the config lines as produced by the original device in its own generated config file and can also cover other config variants if the dictionary is expected to help interactively with processing them.
- Optional sections can be enclosed in square brackets []
- Arguments can be enclosed in <:> by including the argument name inside <:> and followed by valid argument syntax after ":". Example (must not start with any spaces):

```
vlan ports <ports> pvid <pvid:1-4094> [filter-untagged-frame <filtUntag:enable,disable>]
```

Dictionary command argument syntax:

- **<name>:** Argument variable which will accepts any string
- **<name:1-10>:** Argument variable which will accept a number between 1 and 10
- **<name:1-10,>:** Argument variable which will accept a list and/or range of numbers where all numbers must be between 1 and 10
- **<name:value1,value2,etc>:** Argument variable which will accept only discrete values: "value1" or "value2" or ect..
- **<port>:** When name = 'port', the argument variable only accepts a single valid port for the connected host (or in the defined input port range)
- **<ports>:** When name = 'ports', the argument variable accepts a list and/or range of valid ports for the connected host (or in the defined input port range)

Translation lines must follow each dictionary command, and must be indented (start with space or tab). These lines always have 2 fields, in one of these 2 formats:

- <condition_field> = <translation for dictionary command if condition_field is true>
- <condition_field> = &<instruction> [<input based on instruction; can be in double quotes>]

The condition field can contain any of Control::CLI:Extreme attributes in {} brackets. You can find available attributes here: https://metacpan.org/pod/Control::CLI::Extreme#Main-I/O-Object-Methods see: attribute() - Return device attribute value.

The condition field can also contain the <argument> values entered by user in the dictionary command. The condition field is evaluated as a regular perl expression, after making the above {attribute} & <argument> replacements. Condition fields are evaluated in order, until one evaluates to true. Once a condition field evaluates to true, the dictionary command is translated accordingly. If no condition field evaluates to true, then you get a message on the terminal indicating no translation was found for the command.

The translation command is the actual command which ACLI will send to the connected switch if the condition_field evaluates to true. The <argument> values can of course be embedded in the command supplied here. If dealing with an optional <argument> this should again be encloded in square brackets '[]' which can also include a portion of the final CLI command. Also, if using logical operators to verify the setting of an optional <argument> it is best to first assign the <argument> to a variable and then use that variable in the logical operators; this is because an empty variable will be replaced with empty quotes ("), whereas a non-set <argument> will be replaced with nothing.

In the first syntax above, you can chain multiple commands to send to the switch with semicolons (;) and you can also separate these commands over multiple lines provided that every line begins with one or more space/tab characters and the first non-space character is a semicolon (;) followed by a command.

It is also possible to request alternative actions using the &<instruction> format. The following instructions are supported:

- **&ignore ["optional text to print"] :** Do not send the command to the connected host; optionally print a message instead
- **&error ["optional text to print"] :** Stop sourcing and optionally print a message to alert user to a problem
- **&same :** For some target product families, the command is the same and requires no translation. This instruction can also be part of a larger translation script, i.e. within a semicolon list of commands supplied in the first syntax above.

Dictionary Variables

ACLI reserves a special context for dictionary variables. Any variable set inside the dictionary file script, or set by any translation of a dictionary command will be saved in the dictionary scope. These variables can easily be inspected using the new *@vars show dictionary* embedded command (but will remain hidden and not visible when invoking *@vars*, *@vars show* or *@$*):

```
VSP-8284XSQ:1#{ERS}% @vars show dictionary

$dct_DefaultVlan   = 4048
$dct_UniMode       = 1

VSP-8284XSQ:1#{ERS}%
```

These variables can be used like any other ACLI variable, but will typically only be used and called by dictionary translation commands.

The same global name space is however used for all ACLI variables. Which means a variable name is unique and can be "tagged" as used by the dictionary functionality or not.

In order to clearly distinguish dictionary variables (from regular variables), it is best to give them a well defined prefix. For this reason a *@my <prefix>* can be included in the script section of the dictionary file itself. The supplied ERS dictionary file has this statement:

```
@my $dct_*
```

If a *@my* scope was defined in the dictionary file, then dictionary variables can be written inside the dictionary file using the shorthand notation *$*name*. When the dictionary file is parsed and loaded, any variables in that format will automatically be converted to *$<my-prefix>name*. Hence our *$*name* variable will become *$dct_name*.

Using Dictionary Files

In this section we are going to see how dictionary files can be used in practice. We will use the ERS dictionary supplied with ACLI.

We will use an ACLI connection into the VOSS VM, which emulates a VSP8242XSQ. When the ERS dictionary file is first loaded we will get the following:

```
VSP-8284XSQ:1#% @dictionary load ers
Loading dictionary file: C:\Users\lstevens\Scripts\acli\ers.dict

Sourcing ERS Dictionary script

Which VSP UNI config will apply for dictionary translations; CVLAN-UNI (1) or Switched-UN
VSP-8284XSQ:1#{ERS}%
```

There is a huge difference on how VLANs are configured on VOSS ports between CVLAN-UNI and Flex-UNI Switched-UNI. The script embedded in the ERS dictionary file gets executed when the dictionary file is loaded. Currently that script checks to see whether the connected VOSS switch is a DVR Leaf, and if so, then will automatically assume the mode to use will be Switched-UNI. However if the switch is not a DVR Leaf, there is no way to know which mode should be used, hence the user is asked to choose. In these examples we shall choose the simpler CVLAN-UNI option.

```
VSP-8284XSQ:1#{ERS}% @vars show dictionary

$dct_DefaultVlan   = 4048
$dct_UniMode       = 1

VSP-8284XSQ:1#{ERS}%
```

Inspection of the dictionary variables after loading the ERS dictionary will show that a couple of variables have been set. The UNI mode, detected or selected by the user, has been stored in one variable. And another variable has been set with whatever default VLAN is in use on the connected VSP (this could be 1 or 4048, depending on which default mode the VSP was booted into).

In parallel we will also use a pseudo terminal emulating an XOS switch:

```
PSEUDO#% @pseudo list
Available Saved Pseudo Terminals:

   Name                  Origin    Family Type   Port Range
   --------------------  -------   -----------   ----------
   5520-48W-VIM-EXOS     private   ExtremeXOS    1-52
   5520-48W-VIM-VOSS     private   PassportERS   1/1-1/48,2/1-2/4
   X460G2                private   ExtremeXOS

PSEUDO#% @pseudo load 5520-48W-VIM-EXOS
Loading var file C:\Users\lstevens\.acli\.vars\pseudo.5520-48W-VIM-EXOS.vars

5520-48W-EXOS#% @pseudo info

        Pseudo Terminal      : enabled
        Pseudo Name/Id       : 5520-48W-VIM-EXOS
        Pseudo Prompt        : 5520-48W-EXOS#
        Pseudo Command Echo  : disabled
        Pseudo Family Type   : ExtremeXOS
        Pseudo ACLI/NNCLI    : No
        Pseudo Port Range    : 1-52

5520-48W-EXOS#%
```

Note, there is no switch connected in pseudo mode, so using the dictionary functionality here will result in an offline translation without any validation of the correctness of it.

The pseudo port-range is set to match the valid ports of the emulated XOS device and this port range will be enforced when entering dictionary commands with the <port> and <ports> arguments.

```
5520-48W-EXOS#% @dictionary load ers
Loading dictionary file: C:\Users\lstevens\Scripts\acli\ers.dict

5520-48W-EXOS#{ERS}% @echo off output off
Sourcing ERS Dictionary script

5520-48W-EXOS#{ERS}%
```

When the same ERS dictionary file is loaded for an XOS switch, the same embedded script will not request any user input, as on XOS there is no ambiguity on how VLANs can be configured on ports.

We shall now create a couple of VLANs, set the tagging mode on a test port and then set those VLANs on that test port, all done using the same commands which would be used if we were connected to an ERS switch.

We shall first enter configuration context:

```
VSP-8284XSQ:1#{ERS}% config ?
ERS dictionary available syntax
  terminal

VSP-8284XSQ:1#{ERS}% config ?
  network   Configure from a TFTP network host
  terminal  Configure from the terminal
  <cr>

VSP-8284XSQ:1#{ERS}% configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
VSP-8284XSQ:1(config)#{ERS}% █
```

```
5520-48W-EXOS#{ERS}% config ?
ERS dictionary available syntax
  terminal

5520-48W-EXOS#{ERS}% configure terminal
5520-48W-EXOS#{ERS}% █
```

Notice that when partially entering commands and hitting the question mark (?) if the syntax is found valid under the loaded dictionary command, this syntax will be shown and will precede whatever syntax the attached switch might accept. The dictionary syntax is highlighted, to be better visible. In the pseudo terminal case there is no connected switch, so only the dictionary syntax will be offered, if applicable. Hitting the Tab key will also result in the command getting expanded to match avaialble dictionary or connected switch commands. Note that if a valid dictionary command is found, this will trump any equivalent command on the connected switch.

Let us inspect the relevant dictionary translation in the ERS dictionary file:

```
configure terminal
        {is_voss}        = &same          # This definition can also be omitted..
        {is_xos}         = &ignore
```

On a VOSS system, the same command will be passed along, hence the *&same* instruction. Note how the VSP did enter configuration context.

Whereas on an XOS switch there is no configuration context, hence the *&ignore* instruction. So on the XOS switch the command is simply ignored (if a real XOS switch was connected, i.e. not pseudo terminal, then nothing would have been sent to the switch).

The ERS dictionary files can also be used by simply copy-pasting a full ERS config, so any ERS command will need a translation, even for commands which do not map to anything on the connected switch.

Next we shall create a couple of VLANs, using the ERS command syntax:

```
VSP-8284XSQ:1(config)#{ERS}% vlan create ?
      ERS dictionary available syntax
        <vids:2-4094,>

      VSP-8284XSQ:1(config)#{ERS}% vlan create ?
        <2-4059>          Vlan id
      VSP-8284XSQ:1(config)#{ERS}% vlan create
```

Note the 'vlan create' syntax exists on both the ERS dictionary and the connected VSP, alas the former can take a list/range of VLAN ids, whereas the VSP can only take a single VLAN id. Let's enter a VLAN list:

```
VSP-8284XSQ:1(config)#{ERS}% vlan create 10,11 ?
      ERS dictionary available syntax
        type

      VSP-8284XSQ:1(config)#{ERS}% vlan create 10,11 ?
                                                   ^
      % Invalid input detected at '^' marker.
      VSP-8284XSQ:1(config)#{ERS}% vlan create 10,11
```

Note that the connected VSP is not happy anymore with the syntax of what we have entered on the prompt. But the syntax remains valid for the ERS dictionary, so we can continue entering the command (and hitting Tab key to automatically expand command options):

```
VSP-8284XSQ:1(config)#{ERS}% vlan create 10,11 type port ?
      ERS dictionary available syntax
        <cr>
        cist
        <inst:1-8>
        <vvln:voice-vlan>

      VSP-8284XSQ:1(config)#{ERS}% vlan create 10,11 type port ?
                                                              ^
      % Invalid input detected at '^' marker.
      VSP-8284XSQ:1(config)#{ERS}% vlan create 10,11 type port
      VSP-8284XSQ:1(config)#{ERS}% vlan create %s type port-mstprstp 0 &'10,11
      VSP-8284XSQ:1(config)#{ERS}% vlan create 10 type port-mstprstp 0
      VSP-8284XSQ:1(config)#{ERS}% vlan create 11 type port-mstprstp 0
      VSP-8284XSQ:1(config)#{ERS}%
```

We entered the command *'vlan create 10,11 type port'* and the resulting dictionary translation provided the necessary command (or script) to create our VLAN list.

Likewise on our XOS pseudo terminal:

```
      5520-48W-EXOS#{ERS}% vlan create 10,11 type port ?
      ERS dictionary available syntax
        <cr>
        cist
        <vvln:voice-vlan>
        <inst:1-8>

      5520-48W-EXOS#{ERS}% vlan create 10,11 type port
            dict{ERS}% create vlan %s &'10,11
      5520-48W-EXOS#{ERS}% create vlan 10
      5520-48W-EXOS#{ERS}% create vlan 11
      5520-48W-EXOS#{ERS}%
```

And if we look at the relevant command in the ERS dictionary file:

```
      vlan create <vids:2-4094,> type port [cist] [<inst:1-8>] [<vvln:voice-vlan>]    # We thro
            {is_voss}      = @if $*UniMode eq "1"
                           ;       vlan create %s type port-mstprstp 0 &'<vids>
                           ; @endif
                           ; @if "<vvln>"; $*VoiceVlan = <vids>; @endif    # <vids> single v
            {is_xos}       = create vlan %s &'<vids>
```

Next we shall set a port a tagged:

```
VSP-8284XSQ:1(config)#{ERS}% vlan ports 1/1 tagging tagAll ?
ERS dictionary available syntax

   filter-untagged-frame

VSP-8284XSQ:1(config)#{ERS}% vlan ports 1/1 tagging tagAll ?


VSP-8284XSQ:1(config)#{ERS}% vlan ports 1/1 tagging tagAll
VSP-8284XSQ:1(config)#{ERS}% vlan ports 1/1 tagging tagAll
VSP-8284XSQ:1(config)#{ERS}%

5520-48W-EXOS#{ERS}% vlan ports 1/1 tagging tagAll

$dct_PortTag{1}    = tagged

5520-48W-EXOS#{ERS}%
```

The same command entered seems to get executed a second time on the VSP. Whereas a variable is simply set on XOS. Let us look at the relevant dictionary entry:

```
vlan ports  tagging tagAll
        {is_voss}       = @if $*UniMode eq "1"  # CVLAN-UNI
                        ;       &same           # VOSS has this ERS command
                        ; @else  # SW-UNI
                        ;       @for $*port &'
                        ;            $*PortTag{$*port} = tagged
                        ;       @endfor
                        ;       vlan members remove $*DefaultVlan
                        ;       interface gigabitEthernet
                        ;           no private-vlan
                        ;           @error disable
                        ;           flex-uni enable
                        ;           @error enable
                        ;       exit
                        ; @endif
        {is_xos}        = @for $*port &'
                        ;       $*PortTag{$*port} = tagged
                        ; @endfor
```

In the VSP case, the *&same* instruction results in the same ERS dictionary command being re-used, since VOSS has the same syntax available. In the XOS case, there is no way to configure an XOS port for Q-tagging. Only when VLANs are added to an XOS port is the tagging mode set. So we simply have to remember that when adding to this port we will have to tag those VLANs, so we set a variable to remember that.

And so we shall add our VLANs 10 & 11 onto our tagged port 1/1:

```
VSP-8284XSQ:1(config)#{ERS}% vlan members add 10,11 1/1
VSP-8284XSQ:1(config)#{ERS}% vlan members remove $dct_DefaultVlan 1/1
                 vars{ERS}% vlan members remove 4048 1/1
VSP-8284XSQ:1(config)#{ERS}% interface gigabitEthernet 1/1
VSP-8284XSQ:1(config-if)#{ERS}% no private-vlan
VSP-8284XSQ:1(config-if)#{ERS}% exit
VSP-8284XSQ:1(config)#{ERS}% vlan members add %s 1/1 &'10,11
VSP-8284XSQ:1(config)#{ERS}% vlan members add 10 1/1
VSP-8284XSQ:1(config)#{ERS}% vlan members add 11 1/1
VSP-8284XSQ:1(config)#{ERS}%

5520-48W-EXOS#{ERS}% vlan members add 10,11 1/1
5520-48W-EXOS#{ERS}% configure vlan $dct_vlan add ports $dct_port tagged
          vars{ERS}% configure vlan 10 add ports 1 tagged
5520-48W-EXOS#{ERS}% configure vlan $dct_vlan add ports $dct_port tagged
          vars{ERS}% configure vlan 11 add ports 1 tagged
5520-48W-EXOS#{ERS}%
```

The first command was entered, the following commands happened on their own (dictionary command translated to a script). In the XOS case note how both VLANs are added as 'tagged' on our port 1/1. This is correct since we configured that port earlier as a 'tagAll' port.

The relevant dictionary entry (partially trimmed) follows:

```
vlan members [add] <_-4094_>
        {is_voss}       = @if $*UniMode eq "1"  # CVLAN-UNI
                        ;       @if $*DefaultVlan == 4048
                        ;               vlan members remove $*DefaultVlan
                        ;               interface gigabitEthernet
                        ;                   no private-vlan
                        ;               exit
                        ;       @endif
                        ;       vlan members add %s  &'
                        ; @else  # SW-UNI
                                [...omitted...]
                        ; @endif
        {is_xos}        = @for $*vlan &'
                        ;       @for $*port &'
                        ;               @if $*PortTag{$*port} eq 'tagged'
                        ;                       configure vlan $*vlan add ports $*port ta
                        ;               @elsif $*PortTag{$*port} eq 'untagPvid'
                        ;                       @for $*port &'
                        ;                               @if !$*PortPvid{$*port}
                        ;                                       @printf "Please configure
                        ;                                       @exit
                        ;                               @endif
                        ;                               @if $*PortPvid{$*port} eq $*vlan
                        ;                                       configure vlan $*vlan add
                        ;                               @else
                        ;                                       configure vlan $*vlan add
                        ;                               @endif
                        ;                       @endfor
                        ;               @else  # We assume 'untagged' even if this was no
                        ;                       configure vlan $*vlan add ports $*port un
                        ;               @endif
                        ;       @endfor
                        ; @endfor
```

There are a couple of points to note here. For both VOSS and XOS the translation is an ACLI script. However when we executed the commands above, the only commands which became visible were the actual config commands sent to the switch (or pseudo terminal); the *@if*, *@else*, *@endif*, etc.. embedded commands were not echoed at all, which helps reduce the output clutter and gives a better idea of what the translation actually is. This behaviour is achieved via two settings.

The first one is the dictionary echo mode, which by default is set to *'single'*:

```
VSP-8284XSQ:1(config)#{ERS}% @dictionary echo ?
Syntax: @dictionary echo always|disable|single
```

Other ACLI functions, such as alias, variables, history, also have an echo mode, but this is either *'enable'* or *'disable'*. For dictionary echo mode we have:

- **always:** Dictionary command translation is always echoed
- **disable:** Dictionary command translation is never echoed
- **single:** Dictionary command translation is only echoed if the translation is a single command; it is not echoed if the translation is a semicolon fragmented list of commands (i.e. a script, which would be quite lengthy to print out and would wrap over several lines)

The second one is a new setting for the *@echo* embedded command: *@echo sent*

```
VSP-8284XSQ:1(config)#{ERS}% @echo ?
Syntax: @echo info|off|on|sent

VSP-8284XSQ:1(config)#{ERS}% @echo info

Echo of commands & prompts : sent
Echo of command output     : on
```

```
VSP-8284XSQ:1(config)#{ERS}%
```

When the *@echo* mode is off, and ACLI is in scripting/sourcing mode, then commads sent are not echoed to the terminal. In the new *'sent'* mode, a distinction is made between real commands which get sent to the connected device (these are echoed) whereas embedded commands (like flow control embedded commands *@if*, *@else*, *@endif*, etc..) are not echoed. The ERS dictionary file embedded script will automatically enable *@echo sent*

Dictionary Port-ranges

In the previous section we have seen how the ERS dictionary allows us to accept and translate ERS config commands on either a VSP or an XOS. However when entering commands which configure ports, the port number supplied was always a valid port number on the connected VOSS or XOS device. This becomes a challenge if one wants to simply copy-paste a configuration snippet from an ERS switch (which will not have slot-based port numbers) or a stack (which could have port numbers across slots 1 to 8, depending on stack size), since the ERS port numbers might not directly map to the target VSP or XOS switch.

For this reason it is possible to configure the dictionary functionality with the accepted input port-range to use with dictionary translations, using the *@dictionary port-range* embedded command.

For example, imagine we wanted to convert an ERS4950GTS standalone config and we know that such an ERS has 1-48 copper ports and 49-50 as SFP+ ports.

```
VSP-8284XSQ:1(config)#{ERS}% @dictionary port-range input 1-50

Input Port Range  : 1-50 (50 ports)
Mapped Host Ports : 1/1-1/42,2/1-2/8 (50 ports)
Unused Host Ports : 2/9-2/42 (34 ports)

VSP-8284XSQ:1(config)#{ERS}%
```

On our VSP, setting an input port-range to 1-50, immediately maps those ports to the 1st 50 ports available on the connected VSP. In this case we are using the VOSS VM, which is a VSP8284XSQ which has 42 ports acrosss 2 slots. The actual port mapping can be displayed with *@dictionary port-range info*:

```
VSP-8284XSQ:1(config)#{ERS}% @dictionary port-range info

Input Port Range  : 1-50 (50 ports)
Mapped Host Ports : 1/1-1/42,2/1-2/8 (50 ports)
Unused Host Ports : 2/9-2/42 (34 ports)
Mapping detail    :
                     1 => 1/1
                     2 => 1/2
                     3 => 1/3
                     4 => 1/4
                     5 => 1/5
                     6 => 1/6
[...]
                    40 => 1/40
                    41 => 1/41
                    42 => 1/42
                    43 => 2/1
                    44 => 2/2
                    45 => 2/3
                    46 => 2/4
                    47 => 2/5
                    48 => 2/6
                    49 => 2/7
                    50 => 2/8

VSP-8284XSQ:1(config)#{ERS}%
```

Now, if we were to paste an ERS config command, with the ERS original port numbers, we would get this:

```
VSP-8284XSQ:1(config)#{ERS}% vlan ports 43 tagging tagAll
VSP-8284XSQ:1(config)#{ERS}% vlan ports 2/1 tagging tagAll
VSP-8284XSQ:1(config)#{ERS}%
```

Notice how port 43 was automatically converted to 2/1.

In the case of an ERS stack, let us imagine we have an ERS stack of 4 config. So we can modify our XOS pseude terminal to build an equivalent stack of 4 5520 units:

```
5520-48W-EXOS#{ERS}% @pseudo port-range 1/1-52,2/1-52,3/1-52,4/1-52

Port Range: 1:1-52,2:1-52,3:1-52,4:1-52

5520-48W-EXOS#{ERS}% @pseudo info

        Pseudo Terminal    : enabled
        Pseudo Name/Id     : 5520-48W-VIM-EXOS
        Pseudo Prompt      : 5520-48W-EXOS#
        Pseudo Command Echo : disabled
        Pseudo Family Type  : ExtremeXOS
        Pseudo ACLI/NNCLI   : No
        Pseudo Port Range   : 1:1-52,2:1-52,3:1-52,4:1-52

5520-48W-EXOS#{ERS}%
```

Now, we can simply map in the ERS 4 unit stack port-range:

```
5520-48W-EXOS#{ERS}% @dictionary port-range input 1/1-50,2/1-50,3/1-50,4/1-50

Input Port Range  : 1:1-50,2:1-50,3:1-50,4:1-50 (200 ports)
Mapped Host Ports : 1:1-52,2:1-52,3:1-52,4:1-44 (200 ports)
Unused Host Ports : 4:45-52 (8 ports)

5520-48W-EXOS#{ERS}%
```

But the default mapping is not in synch, because an ERS4950 has 48+2 ports whereas a 5520+VIM has 48+4. So assuming we would want the ERS uplink ports 49&50 to be in sych with the 5520 last 2 VIM ports, we now have to rejig the dictionary output mapping port-range like this:

```
5520-48W-EXOS#{ERS}% @dictionary port-range mapping 1:1-48,1:51-52,2:1-48,2:51-52,3:1-48,

Input Port Range  : 1:1-50,2:1-50,3:1-50,4:1-50 (200 ports)
Mapped Host Ports : 1:1-48,1:51-52,2:1-48,2:51-52,3:1-48,3:51-52,4:1-48,4:51-52 (200 port
Unused Host Ports : 1:49-50,2:49-50,3:49-50,4:49-50 (8 ports)

5520-48W-EXOS#{ERS}%
```

The above approach works ok on XOS, which does support stacks like the ERS did. However it won't help if the ERS stack config needs to be converted to XOS smaller stacks or standalone switch, or to VOSS which simply cannot satck. In this case the approach to use is to parse the same ERS stack config several times, once for every subset of ports we need to convert. For example, lets assume we still have an ERS4950 4-unit stack config, and we want to convert this to 4 separate 5520 running VOSS.

We here use a pseudo terminal set to a 5520 in VOSS mode with a 4-port VIM fitted:

```
5520-24W#% @pseudo info

        Pseudo Terminal    : enabled
        Pseudo Name/Id     : 5520-48W-VIM-VOSS
        Pseudo Prompt      : 5520-24W#
        Pseudo Command Echo : disabled
        Pseudo Family Type  : PassportERS
        Pseudo ACLI/NNCLI   : Yes
        Pseudo Port Range   : 1/1-1/48,2/1-2/4

5520-24W#%
```

Let's assume that we will start by mapping ERS unit 4 ports to start with:

```
5520-24W#{ERS}% @dictionary port-range input 4/1-50

Input Port Range  : 4/1-4/50 (50 ports)
Mapped Host Ports : 1/1-1/48,2/1-2/2 (50 ports)
```

```
        Unused Host Ports : 2/3-2/4 (2 ports)

        5520-24W#{ERS}%
```

So we start copy-pasting the ERS port config snippet into the ACLI terminal and we will get only a translation for those slot 4 ports:

```
        5520-24W#{ERS}% vlan ports 1/49-50,4/49-50 tagging tagAll
        5520-24W#{ERS}% vlan ports 2/1-2/2 tagging tagAll
        5520-24W#{ERS}%
```

Notice how only the slot 4 ports get converted. And if we did have some commands which configure ports completely outside of the accepted input port-range, those commads wil lbe simply ignored (with a warning message):

```
        5520-24W#{ERS}% vlan ports 1/1-5,2/1-5 tagging unTagPvidOnly
        Ignoring dictionary command due to empty  after applying dictionary input port-range
        5520-24W#{ERS}%
```

So we basically will have to paste the same ERS port config snippet, 4 times, one for every slot we need to convert, each time specifying the appropriate input port-range to map.

Converting configs with a loaded Dictionary

The ACLI dictionaries can be used by entering input ERS commands one at a time. But can also be used to convert larger config snippets in one shot, and then recovering the translated commands. There are two ways to source the input configs: either using the embedded *@source* command or simply copy-pasting the sections to translate into the ACLI terminal.

If any errors are encountered during the translation, the ACLI sourcing mode will immediately pause, as usual. This will allow you to correct/adjust the failed command and then *@resume* execution from exactly where it had stopped. If using a pseudo terminal, no errors will be seen with the conversions (as no real switch is connected to validate the translated commands). However, if a dictionary file is loaded and sourcing commands, the pseudo terminal will halt execution if it reaches a command which did not have any translation in the dictionary file. This can help spot commands not covered by the dictionary file, even if in pseudo mode.

So the general approach will be the following:

1. Load the dictionary file: *@dictionary load <name>*
2. Set the valid input port-range to convert: *@dictionary port-range*
3. Clear the no errors history: *@history clear no-error-device*
4. Either *@source* the config file, or copy-paste the snippets to convert directly into ACLI
5. If errors are encountered, the sourcing will immediately stop; in this case correct the errors and *@resume* the sourcing
6. Once all commands have been converted, capture to file the translated config using: *@history no-error-device > <filename>*, as in example below

```
5520-24W#{ERS}% @pwd

Working directory is:
C:\Users\lstevens\Scripts\acli\working-dir

5520-24W#{ERS}% @history no-error-device > converted.cfg

acli-dev.pl: Saving output .done
acli-dev.pl: Output saved to:
C:\Users\lstevens\Scripts\acli\working-dir\converted.cfg

5520-24W#{ERS}%
```

ACLI Update

The ACLI terminal and associated utilities covered in this section can be updated to the latest available versions by launching the ACLI update script.

To launch the update script simply use the shortcut under Start / ACLI / ACLI Update:
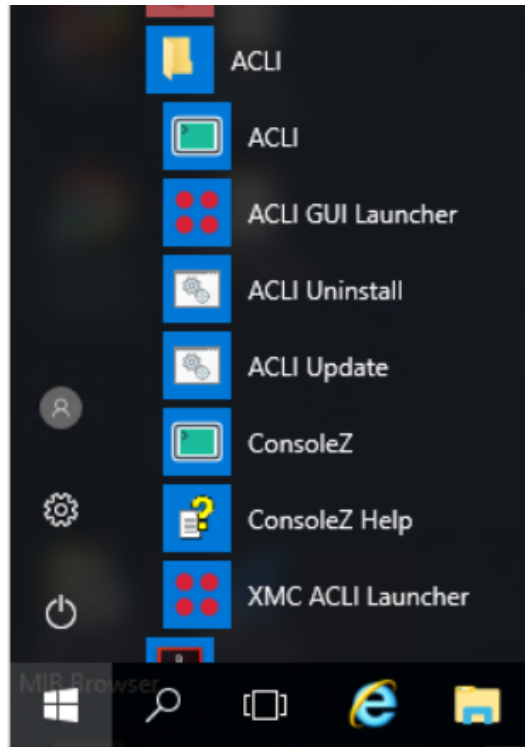
Or alternatively double click the *UPDATE.BAT* file in the ACLI install directory.

You should close all ACLI and ConsoleZ windows before launching the update script. This is not strictly necessary for all updates, but is mandatory if the update script has a newer version of the ConsoleZ executable. The update script will tell you if it cannot perform the update if a ConsoleZ window is still open.

Note, the update script will automatically restart with admin privileges if it detects that it does not have write access in the install directory

The install script will automatically connect to a few URLs. Some of these URLs are internal to Extreme corporate network, but at least one of the URLs will be available on the Internet. The script is menu driven inside a DOS box.

If your versions are up to date, the script will report the following:

```
==========================
ACLI Update script (v1.14)
==========================


Trying http://dante.extremenetworks.com/ACLI-terminal/updates
Trying http://www.oranda.fr/ACLI-terminal/updates .

All files are up to date!

Select desired action:

   (S) - Show full listing of all available update files and versions
   (M) - Provide alternative URL where to pull updates
   (Q) - Quit
```
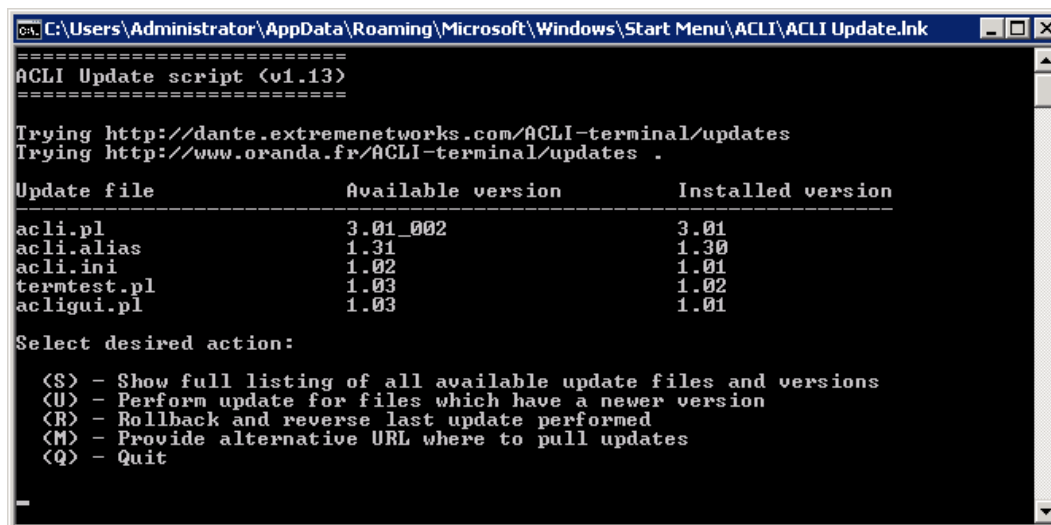
If instead some newer versions are available, then the script will list the newer files available.

To perform the update, simply select the 'U' option. The script will then download the latest versions for the files which need to be updated and will replace those files in your ACLI install directory. The old files however are not deleted, but are placed in a rollback directory. You then simply need to restart ACLI / ConsoleZ and the new versions will be in place. An easy way to see the ACLI version and module versions is to run ACLI and execute the *version* command:

```
ACLI> version
acli.pl version 4.00 (written by Ludovico Stevens)
MSWin32 Perl version 5.026001

Installed Modules used by this script:
        Control::CLI                version 2.07
        Control::CLI::Extreme       version 1.01
        IO::Socket::INET            version 1.35
        IO::Socket::IP              version 0.39
        IO::Select                  version 1.22
```
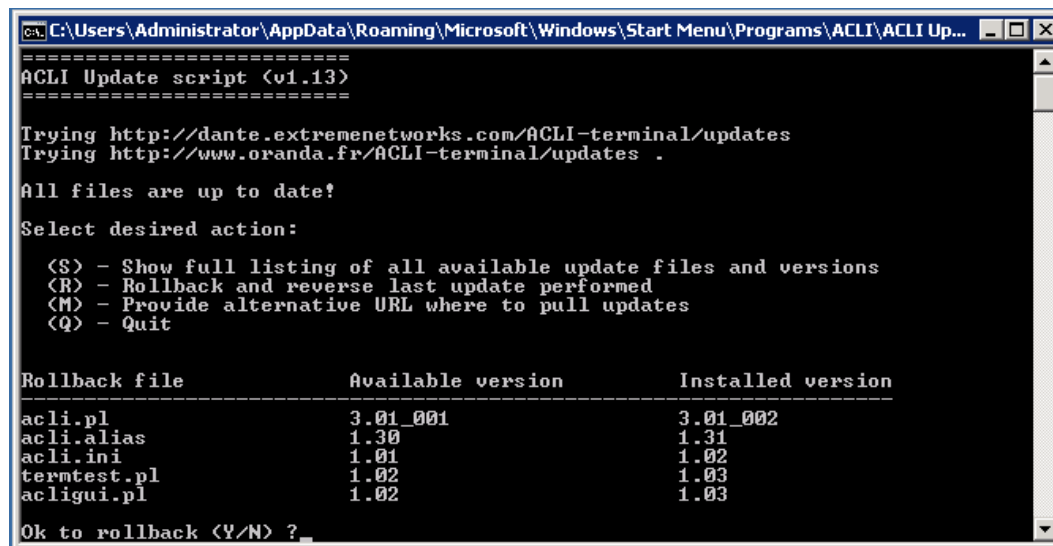
```
            MIME::Base64              version 3.15
            Net::Ping::External      version 0.15
            Net::SSH2                version 0.66
            Net::SSH2 libssh2        version 1.8.0
            Net::Telnet              version 3.04
            Term::ReadKey            version 2.37_01
            Time::HiRes              version 1.9746
            Win32::Console           version 0.10
            Win32::Console::ANSI     version 1.10
            Win32::Process           version 0.16
            Win32::SerialPort        version 0.22
            Win32API::CommPort       version 0.21_001
            Win32API::File           version 0.1203

        ACLI>
```

If for any reason you are not happy with the new versions (maybe a it introduces a new bug! Or it does not work properly), it is possible to roll back to the version you had before the update. To restore those versions simply run the ACLI update script again, and this time select the 'R' option to rollback.

Note that if a new major version of the ACLI distribution is out, it is possible that the update script will not be able to do the update normally. This is usually because the newer version uses a newer Perl version or adds new Perl modules. In this case the update script will still show the newer versions available but will not offer the 'U' update option. Instead it will offer to download the install zip file for the new ACLI distribution. The existing version will then need to be uninstalled and the new version installed in its place.

AFTP script

The AFTP script allows transferring a file to or from many hosts simultaneously using either FTP or SFTP. It can be used to pull the config file from many VOSS switches simultaneously. As the filename will be the same across all the switches, the files will be prepended with the hostname so all obtained files can be stored in the same directory, where they can be edited and modified, and all uploaded back to the same switch again. The script can also be used to push the tgz upgrade software image file to many switches.

The script can be executed from any DOS box or from a ConsoleZ tab window:

```
C:\>aftp
aftp.pl version 1.04

 Simultaneously transfers files to/from 1 or more devices using either FTP or SFTP
 When GETting the same file from many devices, prepends device hostname/IP to filename
 When PUTting the same file back to many devices, only specify the file without prepend

Usage:
 aftp.pl [-l <user>] [-p <path>] <host/IP/list> [<ftp|sftp>] <get|put> <file-list/glob>

 aftp.pl -f <hostfile> [-l <user>] [-p <path>] [<ftp|sftp>] <get|put> <file-list/glob>

 -f <hostfile>     : File containing a list of hostnames/IPs to connect to
 -l <user>         : Use non-default credentials; password will be prompted
 -p <path>         : Path on device
 <host/IP/list>    : Hostname or IP address or list of IP addresses to connect to
 <ftp|sftp>        : Protocol to use; if omitted will default to FTP
 <get|put>         : Whether we get files from device, or we put files to it
 <file-list/glob>  : Filename or glob matching multiple files or space separated list
```

For example, to recover the config.cfg file from several switches you can do:

```
C:\>aftp 10.134.169.91-92,81-84,171-172 ftp get config.cfg
Connecting to hosts:
  1 - 10.134.169.91
  2 - 10.134.169.92
  3 - 10.134.169.81
  4 - 10.134.169.82
  5 - 10.134.169.83
  6 - 10.134.169.84
  7 - 10.134.169.171
  8 - 10.134.169.172

Copying files ...............<5>.<6>.......<8>....<3>.......<1>.<7>.<4>..........<2>


C:\>dir *.cfg
 Volume in drive C is Avaya eSOE
 Volume Serial Number is C87E-793B

 Directory of C:\Users\ludovicostev\Downloads

28/04/2016  22:29            14,796 10.134.169.171_config.cfg
28/04/2016  22:29            14,223 10.134.169.172_config.cfg
28/04/2016  22:29            19,348 10.134.169.81_config.cfg
28/04/2016  22:29            18,347 10.134.169.82_config.cfg
28/04/2016  22:29            18,463 10.134.169.83_config.cfg
28/04/2016  22:29            16,649 10.134.169.84_config.cfg
28/04/2016  22:29            29,930 10.134.169.91_config.cfg
28/04/2016  22:29            37,833 10.134.169.92_config.cfg
10/02/2016  17:06               477 default.cfg
               9 File(s)        170,066 bytes
               0 Dir(s)  367,696,596,992 bytes free
```

Note that if the same file is fetched from more than one switch, then the switch IP address is pre-pended to the file recovered, as shown above. Now you can edit all the above files using your preferred text editor. Once done, you can push the updated files back to their originating switch in one shot like this:

```
C:\>aftp 10.134.169.91-92,81-84,171-172 ftp put config.cfg
Connecting to hosts:
  1 - 10.134.169.91
  2 - 10.134.169.92
  3 - 10.134.169.81
  4 - 10.134.169.82
  5 - 10.134.169.83
  6 - 10.134.169.84
  7 - 10.134.169.171
  8 - 10.134.169.172

Copying files .................<4>..<6><8>......<5>.<3><7><2><1>


C:\>
```

ACMD script

The ACMD script allows bulk execution of a set of CLI commands against many switches, either via SSH or Telnet.

The script can be executed from any DOS box or from a ConsoleZ tab window:

```
C:\>acmd
acmd.pl version 1.05

 Execution of CLI commands/script in bulk to many Extreme Networks devices using SSH or T

Usage:
 acmd.pl [-agiopty] [-l <user>] <host/IP/list> <telnet|ssh> "semicolon-separated-cmds" [<
 acmd.pl [-agiopty] [-l <user>] -s <script-file> <host/IP/list> <telnet|ssh> [<output-fil
 acmd.pl [-agiopty] [-l <user>] -f <hostfile> <telnet|ssh> "semicolon-separated-cmds" [<c
 acmd.pl [-agiopty] [-l <user>] -f <hostfile> -s <script-file> <telnet|ssh> [<output-file
 acmd.pl [-agiopty] [-l <user>] -x <spreadsheet>[:<sheetname>]!<column-label> <telnet|ssh
 acmd.pl [-agiopty] [-l <user>] -x <spreadsheet>[:<sheetname>]!<column-label> -s <script-

  -a                 : In staggered mode (-g) abort further iterations if at least one host
  -f <hostfile>    : File containing a list of hostnames/IPs to connect to; valid lines:
                     :   <IP/hostname>       [<display-name>] [# Comments]
                     :  [<IP/hostname>]:<port> [<display-name>] [# Comments]
  -g <number-N>    : Stagger job over more iterations each for a maximum of N hosts;
                       if not specified, job is performed against all hosts in a single cycl
  -i                 : Create output file per-host, using filename <host/IP>[_<output-file>]
  -l <user>        : Specify user credentials to use (password will be prompted)(default =
  -o                 : Overwrite <output-file>; default is to append
  -p <password>    : Specify a password via command line (instead of being prompted for it
  -s <script-file> : File containing list of commands to be executed against all hosts
  -t <timeout>     : Timeout value in seconds to use (default = 20secs)
  -x <spreadsheet>[:<sheetname>]!<column-label>  : Spreadsheet file (Microsoft Excel, Open
                       Spreadsheet must be a simple table where every row is a device with a
                       of parameters. The first row of the table must be a label for the col
                       The label corresponding to the column with the switch IP/hostnames mu
                       supplied in <column-label>. The other column labels can be embedded a
                       $<label-name> in the supplied CLI commands or script file.
                       The <column-label> and $<label-name> names are case insensitive and a
                       within them in the spreadsheet will be replaced with the '_' undersco
                       The <sheetname> is optional; if not supplied the first sheet of the s
                       will be used
  -y                 : Skip job detailed summary and user confirmation prompt
  <host/IP list>   : List of hostnames or IP addresses
                     : Note that valid IP lists can be written as: 192.168.10.1-10,40-45,51
                     : IPv6 addresses are also supported: 2000:10::1-10 (decimal range 1-10)
                     : 2000:20::01-10 (hex range 1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,10)
  <telnet|ssh>     : Protocol to use
  <output-file>    : Output file (and suffix with -i) for output filenames
```

If you don't use the *-g* flag, the script will attack all nodes in one go; with SSH this will be slow initially as the SSH authentications are blocking calls. This script will run a lot faster with Telnet.

If you use the *-g* flag, you can stagger it to do N switches at a time; with SSH it is best to stagger around 10 switches at a time.

The critical thing when working with many switches is keeping track of when things go wrong (if you can't connect to 1 switch, or some switch does not like one of your commands). The approach taken is that if a command or the connection fails to all switches in the 1st iteration, then the script bombs out => changes were not made on any switch (unless 1 command gave an error across all switches, in which case all preceding commands in your script will have executed). If instead only a few switches fail during an iteration (and you are using the staggered mode with *-g*) you can control whether you would like to carry on with subsequent iterations (default) or not (set the *–a* flag for abort). In any case, if the script succeeds on some switches, but fails on others or is not

executed on others because the last iterations were skipped (-g + -a) the list of all switches for which the script was not executed will be stored in a file *<hostfile>.retry* ; that way you can easily re-trigger the same script for just those switches which are remaining (after you've fixed the problem, whatever that was..). The *<hostfile>.retry* file will also include information about the error which each host failed on. And anyway you get a detailed summary of what the script is setting off to do, and you need to confirm before it gets going (unless you force an immediate start with –y)

The *<hostfile>* can take the same syntax as an IP hosts file; for example:

```
10.134.161.41
10.134.161.42
10.134.161.43
10.134.161.44
10.134.161.81    vsp8000-1
10.134.161.82    vsp8000-2
```

So you can list just the IP addresses or hostnames; in the former case, you can provide a switch name in the same file, as in example above for .81 & .82, then these switches will be referred to by name in output dialogue. Also if a switch name is provided, it will replace any occurrences of $$ in the CLI script.

A sample *<script-file>* file, which sets the SNMP location/contact and changes the CLI passwords for 'ro' user:

```
config term
snmp-server location "test test"
snmp-server contact "Ludovico"
username ro level ro // ro // newro // newro
show cli password
end
```

Note that the same ACLI syntax using *//* can be used to feed data to commands.

This is the output of it running:

```
C:\>acmd.pl -o -f myhosts -s myscript ssh output.log
==============================================================================
Identified 33 hosts to run job against
Job consists of pushing CLI script contained in file: myscript
Performing job over single iteration
-> job will be performed by connecting to all 33 hosts at the same time
SSH will be used with default credentials: rwa/rwa
Any output received from hosts will be collected in file: output.log
-> output file 'output.log' already exists and will be overwritten!
If the script succeeds on some hosts but fails on others
-> list of hosts which failed will be listed in file: myhosts.retry
-> file 'myhosts.retry' already exists and will be overwritten!
==============================================================================
OK to proceed [Y = yes; any other key = no] ? y
Connecting to 33 hosts .................................................<7>.<10><12>..
Entering PrivExec on 33 hosts <6>....<19>..<26>.<33> done!
Executing CLI script on 33 hosts
- config term   ...<6>.<13>.<15>.<21>.<33> done!
- snmp-server location "test test"   ...<7>.<11>..<19>.<21>..<23>.<30>.<32>..............
- snmp-server contact "Ludovico"   ....<8>.<13>.<19>.<22>.<27>.<28>.<33> done!
- end   ...<7>.<13>.<14>.<21>.<33> done!
Disconnecting from 33 hosts
Output saved to file output.log
```

Using $$ variable:

```
acmd -l rwa -p rwa -f hosts.txt ssh "enable; config term; prompt $$; router isis; sys-nam
```

And <script-file> has:

```
10.7.6.8   BEB-608
10.7.6.9   BEB-609
10.7.6.14  BEB-614
```

```
10.7.6.15  BEB-615
10.7.6.20  BEB-620
10.7.6.21  BEB-621
```

Using a spreadsheet file (xls, xlsx, xlsm, csv, ods, sxc):


```
C:\Users\lstevens\Scripts\acli\working-dir>acmd -l rwa -p rwa -x excel.csv!Switch ssh "en
================================================================================
Identified 2 hosts to run job against
Job consists of pushing CLI commands provided in command line (5 commands)
Performing job over single iteration
-> job will be performed by connecting to all 2 hosts at the same time
SSH will be used with 'rwa' username and password provided
Any output received from hosts will be discarded (config only script)
If the script succeeds on some hosts but fails on others
-> list of hosts which failed will be listed in file: acmd.retry
-> file 'acmd.retry' already exists and will be overwritten!
================================================================================
OK to proceed [Y = yes; any other key = no] ? y
Connecting to 2 hosts ..............................<1>.<2> done!
Entering PrivExec on 2 hosts ..<2> done!
Executing CLI script on 2 hosts
- config term   ..<2> done!
- prompt $name   ..<2> done!
- router isis   ..<2> done!
- sys-name $name   ..<2> done!
Disconnecting from 2 hosts
```

ACLI GUI script

This tool is a helper to allow you to launch ACLI tabs against a shorthand list of IP addresses (or from a *-f* hosts file) without having to manually open a new tab in the ACLI window and *'open'* against each IP address. At the same time the tabs will be named using the IP address (or the switch name, if this was provided in the *-f* hosts file). This script has a GUI window which will launch if only partial information is provided on the command line (or always if the *-g* switch is set).

A shortcut for ACLI GUI is included in the Start / ACLI shortcuts menu:
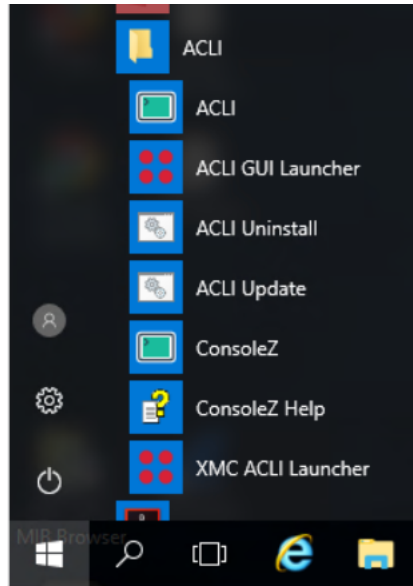
The script can also be executed from any DOS box or from a ConsoleZ tab window:

```
C:\>acligui -h
acligui.pl version 1.12

Usage:
 acligui.pl [-gimnpstuw] [<hostname/IP list>]
 acligui.pl [-gimnpstuw] -f <hostfile>

 <host/IP list>   : List of hostnames or IP addresses
                  : Note that valid IP lists can be written as: 192.168.10.1-10,40-45,51
                  : IPv6 addresses are also supported: 2000:10::1-10 (decimal range 1-10)
                  : 2000:20::01-10 (hex range 1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,10)
                  : As well as IP:Port ranges: [<hostname/IPv4/IPv6>]:20000-20010
 -f <hostfile>    : File containing a list of hostnames/IPs to connect to; valid lines:
                  :   <IP/hostname>          [<name-for-ACLI-tab>] [-n|-t] [# Comments]
                  :   [<IP/hostname>]:<port> [<name-for-ACLI-tab>] [-n|-t] [# Comments]
                  : The -n or -t flags will be passed onto ACLI when connecting to that h
 -g               : Show GUI even if host/IP and credentials provided
 -h               : Help and usage (this output)
 -i <log-dir>     : Path to use when logging to file
 -m <script>      : Once connected execute script (if no path included will use @run sear
 -n               : Launch terminals in transparent mode (no auto-detect & interact)
 -p ssh|telnet    : Protocol to use; can be either SSH or Telnet (case insensitive)
 -s <sockets>     : List of socket names for terminals to listen on
 -t <window-title>: Sets the containing window title into which all connections will be c
 -u user[:<pwd>]  : Specify username[& password] to use
 -w <work-dir>    : Working directory to use (including for <hostfile>)
```

If for example you just execute ACLI GUI without any arguments this will launch the ACLI GUI Launcher window (for which you should also have a shortcut under Start / ACLI). The window also allows you to set all the same arguments (IP address list, username, password, SSH or Telnet, working and logging directories, socket names and run script) that you can specify via the command line. And if you did specify some options via the command line, these will automatically appear as pre-populated once the window is opened.
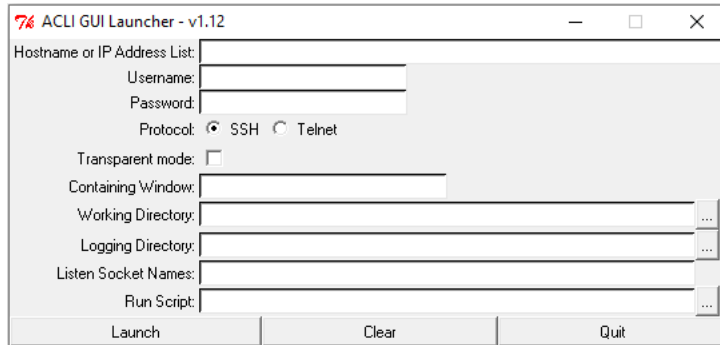
The window has the following input dialogues:

- **Hostname or IP address List**: Can take a list of hostnames or IP addresses; in the case of IP addresses lists can be provided in compact form, e.g. 192.168.10.10-15,22,24
- **Username**: Username which will be used to connect to all switches.
- **Password**: Password which will be used to connect to all switches.
- **Protocol**: Select either SSH or Telnet (default is SSH)
- **Transparent mode**: If set, the ACLI instances will not go into Interactive mode, but will remain in Transparent mode.
- **Containing Window**: If set, this will determine the title of the ACLI window where the ACLI sessions will be opened; if such a window is already open, the newly launched ACLI session tabs will appear in that window. If not set, the ACLI sessions will appear in a generic window named *"ACLI terminal launched sessions"*. This input box also has a pull-down, offering a history of values entered in this box.
- **Working Directory**: Working directory to set on ACLI sessions once they are launched
- **Logging Directory**: Logging directory to use on ACLI sessions once they are launched
- **Listen Socket Names**: Optional list of socket names the launched ACLI sessions should listen to
- **Run Script**: Optional run script to immediately execute against switch once the ACLI session is launched

Here are some examples..

You want to connect to a bunch of switches, for which the IP addresses are 192.168.10.10-15,22,24

```
C:\>acligui 192.168.10.10-15,22,24
```

This will open up the ACLI GUI Launcher window, from where you can populate the username & password fields and then click on "Launch" which will then open 8 ACLI Tabs and connect to each of the switches. Note the shorthand way that IP addresses can be listed.
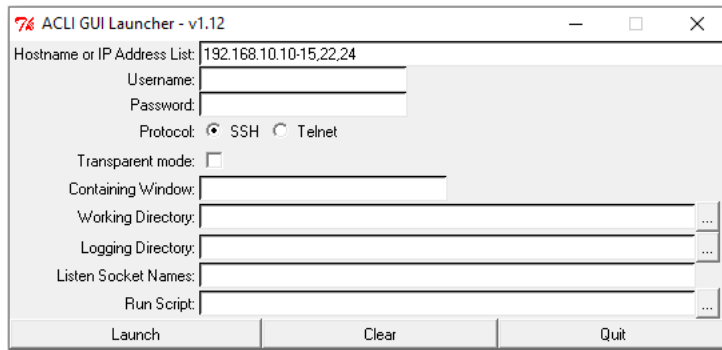
Or you could already specify the username to use:

```
C:\>acligui -u rwa 192.168.10.10-15,22,24
```

In this case the ACLI GUI Launcher window will still open, but now you only need to provide the password and then click "Launch"
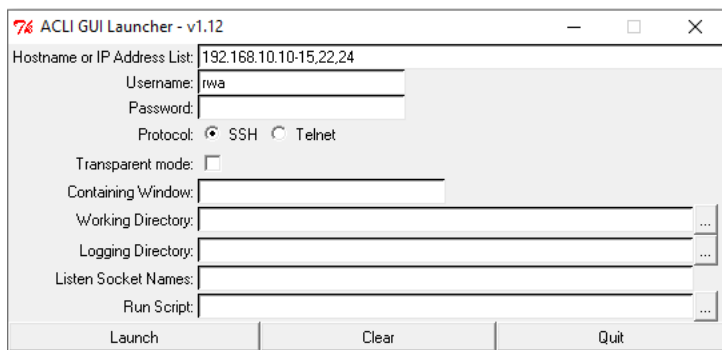
If instead both username and password are provided on the command line:

```
C:\>acligui -u rwa:rwa 192.168.10.10-15,22,24
```

In this case the ACLI GUI Launcher window will not open and you will directly get the desired Console Window with 8 ACLI Tabs each connected to the selected IPs.
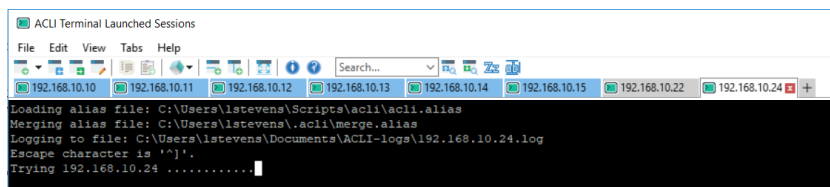
However, in this latter case you had to type the password in clear case on the command line, which may be undesirable.

If you wanted the above to still open the GUI window (so that logging & working directory can be set) then add the -g switch:

```
C:\>acligui -g -u rwa:rwa 192.168.10.10-15,22,24
```
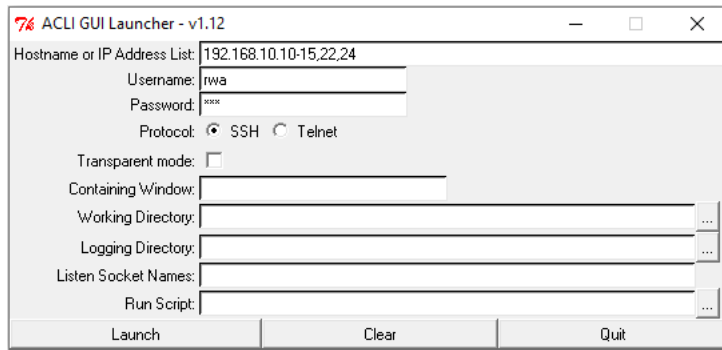
An alternative way to launch ACLI GUI, is via Start / Run, using the same command line syntax as above, but specifying *acligui.vbs* instead of just *acligui*:

```
acligui.vbs -u rwa 192.168.10.10-15,22,24
```

And finally, if you wanted to share an ACLI shortcut to connect to a bunch of switches, you can do the following:

1. Create a batch file (.bat extension) containing:

```
@echo off
acligui.vbs -p ssh -u "<username>[:<password>]" -w "%CD%" -f %0 -t "Window Title"
exit

# List hosts below
<IP-1>                   <Hostname-1>
<IP-2>                   <Hostname-2>
[<IP-3>]:<Port-1>        <Hostname-3>    [-n|-t] [# Comments]
[<IP-3>]:<Port-2>        <Hostname-3>    [-n|-t] [# Comments]
...
```

2. Place the file in the directory you wish to be used as working directory once connected to switches
3. Run the batch file directly, or make a shortcut to it and run that

Notes:

- Always place double quotes around the credentials as shown, in case of special characters
- If the '%' character is present in the credentials, it will need to be escaped by entering it twice '%%'
- Tested with password containing all of these special characters: £$%^&*_+-=<>#/\|[]{}!;:@~
- Always place double quotes around any value containing the space character, as seen above for "Window Title"
- The per entry optional *-t* or *-n* can be provided for terminal server connections where ACLI should not enter interactive mode with *(-n)* or should with *(-t)*

Note that when this tool spawns new ACLI terminal instances it will use the ACLI spawn file *acli.spawn*; see the manual entry for ACLI spawn file.

XMC ACLI script

This tool extracts all discovered devices from Extreme Management Center (XMC) using the GraphQl API and displays them all preserving the XMC Site hierarchy in a tabular output. The user can then browse or filter the entries, make a selection of devices he wishes to connect to with ACLI, then easily launch an ACLI session against all of the selections.

Note that a minimum version of XMC 8.1.2 is required, as this is the first version where XMC added support for the GraphQl interface.

A shortcut for XMC ACLI Launcher is included in the Start / ACLI shortcuts menu:
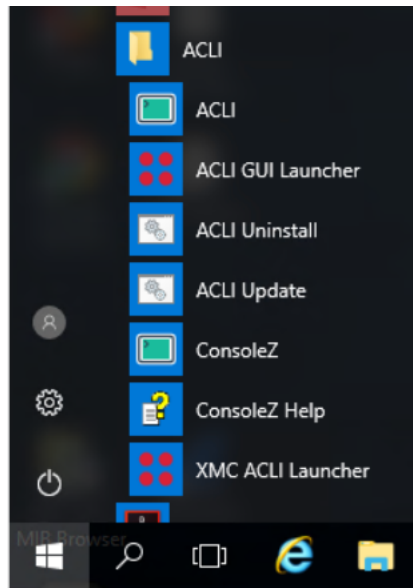
The script can also be executed from any DOS box or from a ConsoleZ tab window:

```
C:\>xmcacli -h
xmcacli.pl version 1.07

Usage:
 xmcacli.pl [-fgimnpqstuw] [<XMC server/IP[:port]>]

 <XMC server/IP[:port]>: Extreme Management Center IP address or hostname & port number
 -f <site-wildcard>    : Filter entries on Site wildcard
 -g <record-grep>      : Filter entries pattern match across any column data
 -h                    : Help and usage (this output)
 -i <log-dir>          : Path to use when logging to file
 -m <script>           : Once connected execute script (if no path included will use @run
 -n                    : Launch terminals in transparent mode (no auto-detect & interact)
 -p ssh|telnet         : Protocol to use; can be either SSH or Telnet (case insensitive)
 -q <graphql-file>     : Override of default xmcacli.graphql file; must be placed in same
 -s <sockets>          : List of socket names for terminals to listen on
 -t <window-title>     : Sets the containing window title into which all connections will
 -u user[:<pwd>]       : Specify XMC username[& password] to use
 -w <work-dir>         : Working directory to use
```

Once run, the script will present the following graphical interface:

To pull all the switch data from XMC, simply enter your XMC-IP:port, XMC username & password, then hit the *"Fetch Data"* button.

The XMC server input box also has a pull down, which will present a list of past successful fetches from XMC servers (useful, if you are working with several XMC installations).

To view the resulting output you may need to re-size the window in order to make the table fit.

The table output can then be filtered down by using the *site filter* & *grep filter* text boxes.

The former has a pull down, which is automatically populated with the XMC site names during fetch time, but remains a text box so any text can be type into it. The latter will do a match on any value in any of the columns; only records with at least a match will then be displayed.

For example, we shall use the site filter pull down to select the XMC site named 'DataCenter':

Which will result in the following output:

Which, if we then wanted to filter down to just the VSP7200s, we can further filter by placing an appropriate matching string in the grep filter box:

Clicking on a column header, will re-arrange all output based on that column; at every click, the sort order will toggle between ascending and descending.

Double-clicking a site folder, auto-selects all entries within. Or you can manually select the entries using the appropriate check box.

In the lower part of the window are some input dialogues:

- **Protocol**: Select either SSH or Telnet (default is SSH)
- **Transparent mode**: If set, the ACLI instances will not go into Interactive mode, but will remain in Transparent mode.
- **Containing Window**: If set, this will determine the title of the ACLI window where the ACLI sessions will be opened; if such a window is already open, the newly launched ACLI session tabs will appear in that window. If not set, the ACLI sessions will appear in a generic window named *"ACLI terminal launched sessions"*. This input box also has a pull-down, offering a history of values entered in this box.
- **Working Directory**: Working directory to set on ACLI sessions once they are launched
- **Logging Directory**: Logging directory to use on ACLI sessions once they are launched
- **Listen Socket Names**: Optional list of socket names the launched ACLI sessions should listen to
- **Run Script**: Optional run script to immediately execute against switch once the ACLI session is launched

The above input dialogues can also be pre-set via the command line, in which case the fields will be pre-populated.

Once the selection of switches is made, and any relevant input dialogues have been set, simply hit the *"Launch"* button to open the ACLI sessions.

Note that there is no need to provide switch credentials, since XMC ACLI Launcher pulls this information from XMC (provided that the admin profile of the device had CLI credentials set).

XMC ACLI Launcher can also be used to quickly launch an SSH/Telnet session against non-Extreme devices (or Extreme devices not yet fully supported in interactive mode with ACLI). What will then happen is that ACLI will still be able to connect and login, but the ACLI session will remain in transparent mode.

There are two files used by XMC ACLI Launcher, which provide formatting of the output as well as the GraphQl query to issue against XMC.

- **xmcacli.graphql**: This file holds the GraphQl query which will be used against the XMC API.
- **xmcacli.ini**: This file holds some initialization paramters as well as the table layout where all the discovered switches will be displayed.

Both files have comments within them with more information about editing them. Both files are located in the ACLI install directory, but note that both files are versioned, which means the ACLI update script may update them if the a newer version is available. If you wish to edit these files it is best to place the modified version under one of the following directories, which will always be inspected before loading the files from the ACLI install directory:

- ENV path *%ACLI%* (if you defined it)
- ENV path *$HOME/.acli* (on Unix systems)
- ENV path *%USERPROFILE%\.acli* (on Windows)

The *xmcacli.ini* file can be edited with the XMC IP/hostname, as well as XMC credentials, so that these fields are pre-populated when XMC ACLI is launched. A *historyDepth* setting also allows to set the depth of the XMC server IP history pull down (which by default is limited to 15 entries).

Otherwise, the main reason for editing these files is to modify the formatting of the tabular display where the switches are displayed. By default the following fields are displayed: *Sysname, IP address, Software version, Location, Up time, Status, Device Model*. Yet, XMC offers a much larger range of information to choose from. You can browse the avaialble keys available by simply pointing your browser to the XMC GraphQl interface:

```
https://<XMC-IP>:<port-number>/nbi/graphiql/index.html
```

The port number is the same port number normally used to access the XMC user interface. It is just the URL which changes.

From this XMC interface, you can simply copy paste the GraphQl query to see what information XMC returns to XMC ACLI Launcher:
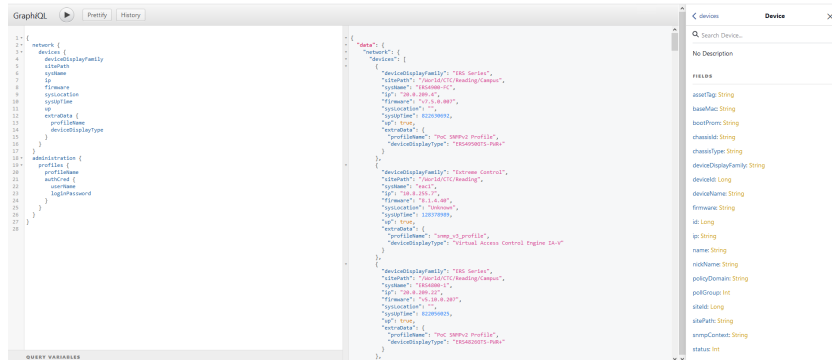
By serching for key *device*, on the right hand side panel, it is possible to see all the other available keys; simply add new keys to the GraphQl query to see what information XMC returns. Note that a lot of extra information is found under the *extraData* key, which unfortunately has been renamed to *deviceData* as of XMC 8.2.0

To add new columns to the XMC ACLI Luncher, it is simply a matter of modifying the *xmcacli.graphql* file to include the extra keys, so that this information will be pulled down from XMC, and to modify the *xmcacli.ini* file so that a new column entry is defined. Entries in the *xmcacli.ini* file need to be in the following format:

```
[<key-name>]
display          = "<name to use in column header>"
type             = <String | Time | Flag | YesNo | DotDecimal | Number> ; indicates how to
```

As an example, if we wanted to add a new colum to show the device's MAC address, we could add the XMC GraphQl key *'macAddress'* to the *xmcacli.graphql* file:

```
{
  network {
    devices {
      deviceDisplayFamily
      sitePath
      sysName
      ip
      firmware
      sysLocation
      sysUpTime
      up
      extraData {
        profileName
        deviceDisplayType
        macAddress   <======
      }
    }
  }
  administration {
    profiles {
      profileName
      authCred {
        userName
        loginPassword
      }
    }
  }
}
```

And then add the following entry into the *xmcacli.ini* file:

```
[macAddress]
display   = MAC Address
type   = String
```

Note that the order in which the entries are placed in *xmcacli.ini*, also determines the order in which the table rows are displayed in XMC ACLI Launcher.

Re-starting XMC ACLI and performing a new *'Fetch Data'* will now include the MAC address information:
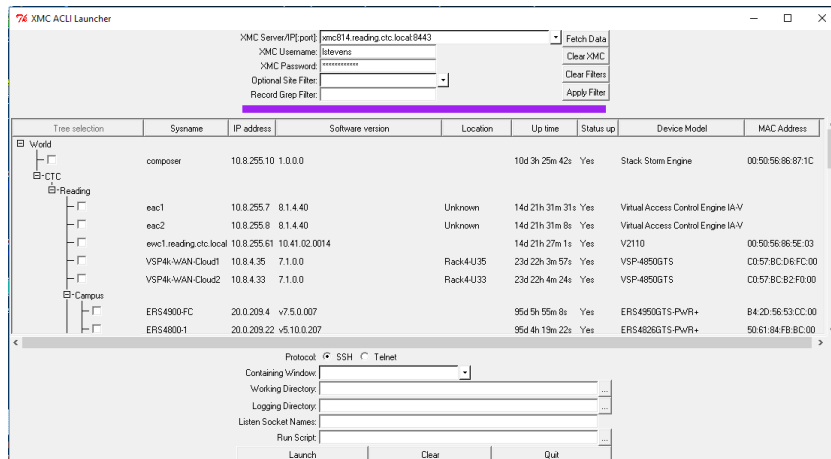


Likewise, if one wanted to remove one of the columns which XMC ACLI Launcher shows by default, it is sufficient to delete (or comment out) the relevant entry in the *xmcacli.ini* file. You can also remove the relevant key from the *xmcacli.graphql* file, but there are some keys which must never be deleted, as these are vital for XMC ACLI Launcher's operation (even if you decide not to display them in the table; these are keys: *sitePath, ip, sysName, profileName, deviceDisplayFamily*

Another reason for editing the *xmcacli.graphql* file is that XMC's early GraphQl versions were a bit in flux initially with the structure of the data they offer. Some key names have changed between XMC 8.1.x and XMC 8.2, like for instance the *extraData* key becomes *deviceData* in 8.2 and the old key name is no longer recognized. Since that key is used by XMC ACLI Launcher, the *xmcacli.graphql* file cannot work for both versions, and the author sees no point in slowing down XMC ACLI in order to check the XMC version first, and then to have a dictionary of inconsistent GraphQl API calls against every XMC version. The default *xmcacli.graphql* file uses the newer *deviceData* key, but if you needed this tool to work with pre-8.2 versions of XMC you will get a similar error:
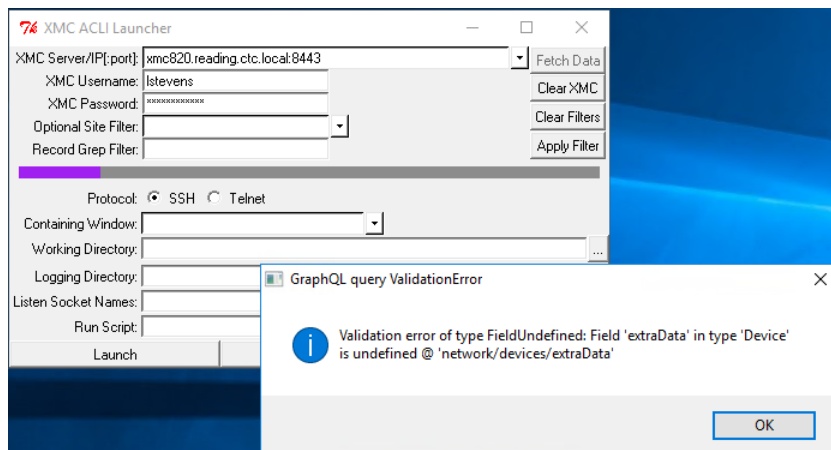
You'll need to go and modify the *xmcacli.graphql* file and set the appropriate GraphQl keys used in your XMC version. In the case where XMC ACLI needs to work against both XMC 8.1.x and XMC 8.2.x, it is possible to create separate shortcuts and for each shortcut specify a different *xmcacli.graphql* file using the command line switch *-q <graphql-file>*.

Note that when this tool spawns new ACLI terminal instances it will use the ACLI spawn file *acli.spawn*; see the manual entry for ACLI spawn file.

# Grep script

The Grep script is provided with the ACLI Windows distribution simply because Windows does not have one.

The Grep script can be executed from any DOS box or from a ConsoleZ tab window:

```
C:\>grep
grep.pl version 2.2

Usage:  [-iv] "pattern" [<file or wildcard>] [<2nd file>] [<3rd file>] [...]

        -i: Case insensitive pattern match
        -v: Return non matching lines

        Pattern syntax for logical operators:
         <str1>and<str2> = "<str1>&<str2>"
         <str1>or<str2>  = "<str1>|<str2>"
        Alternatively, use standard perl pattern match: /pattern/
```

The *pattern* provided can also be a Perl regular expression, in which case it needs to be enclosed with the '/' character.
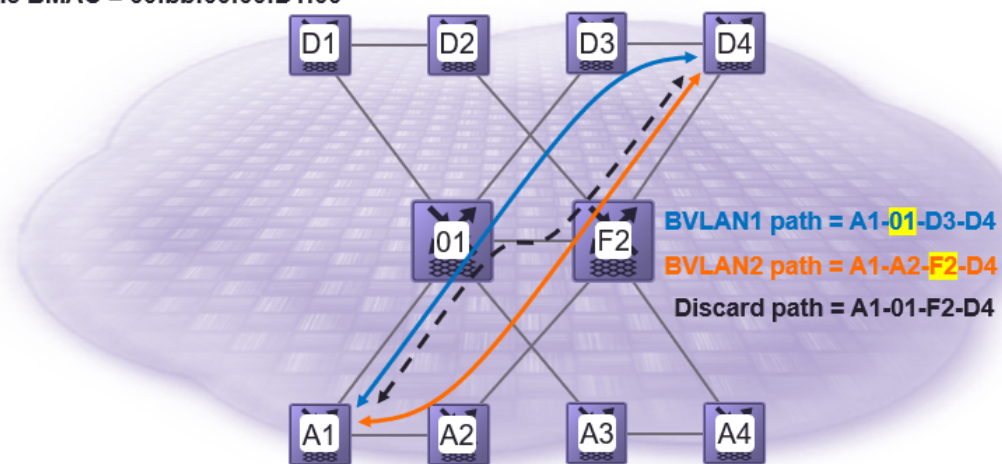
This Grep script has nothing to do with ACLI's grep capability.

Spb-Ect script

The Spb-Ect script can be used to see how SPB allocates available shortest paths to BVLANs.

For example in the following topology we can use the script to identify the paths programmed in the available BVLANs


```
C:\>spb-ect.pl
Specify number of paths to compare[default = 2] :3
Specify number of BVLANs in use[default = 2] :
Comma separated list of nodes on path 1 :a1,01,d3,d4
Comma separated list of nodes on path 2 :a1,a2,f2,d4
Comma separated list of nodes on path 3 :a1,01,f2,d4

Lexicographic ordering done AFTER applying ECT Masks
======================================================

 Processing for BVLAN1, after applying ECT Mask 00
            XORed Path 1 : a1,01,d3,d4
            XORed Path 2 : a1,a2,f2,d4
            XORed Path 3 : a1,01,f2,d4
            Sorted XORed Path 1 : 01,a1,d3,d4 <-- chosen
            Sorted XORed Path 2 : a1,a2,d4,f2
            Sorted XORed Path 3 : 01,a1,d4,f2

 Processing for BVLAN2, after applying ECT Mask ff
            XORed Path 1 : 5e,fe,2c,2b
```

```
                    XORed Path 2 : 5e,5d,0d,2b
                    XORed Path 3 : 5e,fe,0d,2b
                    Sorted XORed Path 1 : 2b,2c,5e,fe
                    Sorted XORed Path 2 : 0d,2b,5d,5e <-- chosen
                    Sorted XORed Path 3 : 0d,2b,5e,fe

SUMMARY
=======
         Path 1 used by BVIDs: 1
         Path 2 used by BVIDs: 2
         Path 3 used by BVIDs:

C:\>
```