# Mass Config Push


- Basics
- Output Grouping
- State of device before Push - require "enable" or "configure" CLI mode
- Command parsing
- Macros
  - Modifiers
  - Actions
- How commands are sent to devices
- Command parsing examples
- Debugging failed pushes - "INTERACTION_ERROR" group


## Basics

Mass Config Push in Unimus is used to send a set of commands to a group (or all) of devices in your network. This allows you to roll out mass config changes across your network, or query command outputs from a large set of devices. Before you can push to devices, each device must be in a "discovered" state (normally done during device addition into Unimus). Config push will not be performed on undiscovered devices, and they will be grouped in the "Un-discovered devices" group.

To perform a push, you first need to create a "Push Preset" from the Mass Config Push home page. For the "Commands", you can provide the same sequence of commands and key-strokes to Unimus as you would when interacting with the device CLI manually.

After a preset is created, you can click  "Run now" to push the provided commands to the selected devices.

## Output Grouping

The commands you specify in the Push Preset are sent to the devices, one line at a time (more info on how Unimus interacts with the device in How commands are sent to devices). The outputs of the executed commands are evaluated on all devices, and a new "Output Group" is created for each unique output during the push. If 2 devices reply with the same output, both devices are assigned to a single output group belonging to that unique output.

As an example, if you push a command to 200 devices, but across all devices there will only be 2 unique outputs to this command, Unimus will create only 2 output groups, and assign each device to the right output group. Output grouping means you don't need to inspect outputs of 200 devices after a config push - saving you time, and making navigation in large push outputs easy.

## State of device before Push - require "enable" or "configure" CLI mode

You have the option of letting Unimus ensure that each device in the Push Preset is in a desired CLI mode before your specified commands are pushed. You can check "Require enable mode" and/or "Require configure mode" in the Push Preset, and Unimus will make sure the device is in the specified mode before executing your commands. Modes available on devices are discovered during Discovery, using mode credentials configured in the "Credentials" screen. You can check which modes Unimus was able to discover on devices using the "Devices > Info" window.

If a device can not be switched into the specified CLI mode before the push, Config Push will NOT push the provided commands to the device, and any devices failing the mode switch will be grouped in appropriate "mode not supported" output groups.

The "enable" and "configure" mode names are only symbolic - each device type has it's own modes. In general, the "enable" mode means the 2nd highest possible CLI mode on the device, and the "configure" mode the 3rd highest possible CLI mode on the device (if the device type supports this). Taking Cisco as an example, the "user exec" mode would be 1st possible login mode, "privilege exec" would be the 2nd (enable) mode, and "configure mode" would be the 3rd (configure).

Please see our Device mode table article for more info on how the modes in the push preset map to various vendors / device types.

## Command parsing

In a Push Preset, you provide a command set to Unimus, which will be parsed for submission modifiers and control sequences, and then sent to the device(s).

If you do not provide any submission modifiers or control sequences, Unimus will split the provided command set line-by-line, and then send it to selected device(s) one-by-one.
Basically, you give Unimus a bunch of commands, and it will distribute them to your network.

For example, specifying push commands like this will work as expected, and Unimus will handle pagination ("-- more --") natively:

```
show version
show ip interface brief
show vlan brief
```

For details on exactly how commands are sent to devices, please see How commands are sent to devices.

# Macros

Since version 2.0.3, Unimus supports command submission modifiers and sending control sequences. Release 2.4.0 introduced overhaul of how modifiers work.

To customize Mass Config Push Presets Unimus supports using of Macros. These are Modifiers, Actions and User variables.

## Modifiers

Modifiers are two-word macros used to adjust the interaction behavior with the remote device. Normally, Unimus waits for some known output (for example a prompt) after each sent command line (see How commands are sent to devices for more info).

Since version 2.4.0 modifiers are used in the following way:

- allowed modifiers: wait-echo, wait-reply, enter, fail-on-error
- format: $(modifier_name yes/no)
- must always be at the end of a line
- multiple consecutive modifiers are allowed
- a single modifier can only be declared once
- are followed by <Enter> unless "$(enter no)" is used

Modifier submission alternatives example:

```
$(wait-reply yes)
$(wait-reply no)
```

Specifying "$(wait-reply no)" will cause Unimus to instantly continue sending the next command line without waiting for any device outputs.

This can be useful when for example the device doesn't respond with any data after sending some command/line, and you want Unimus to continue sending data to the device.

(please check the Command parsing examples section for more details)

## Actions

Actions are two-word macros used to send control sequences and delays for interactions requiring pauses during execution.

Since version 2.4.0 actions are used in the following way:

- allowed actions: send, delay
- format: $(action_name value)
- send inputs a control sequence (it is not followed by <Enter> unless "$(enter yes)" is used)
- delay simply waits for a specified time (in ms)
- can be placed at any position in a line

Action submission example:

```
$(send 0x03)
$(delay 100)
```

As mentioned above, <Enter> is not sent after a control sequence by default, but this can be forced by appending "$[enter]" to the end of the line. Full list of command sequences available in this article: VT100 Control / Command sequences.

(please check the Command parsing examples section for more details)

Both submission modifiers and command sequences support escaping by prefixing them with an additional "\":

```
\$(send 0x03)  = "$(send 0x03)" will be sent literally to the device(s),
instead of a control sequence
\$(enter yes) = "$(enter yes)" will be sent literally to the device(s),
instead of modifying command submission behavior
```

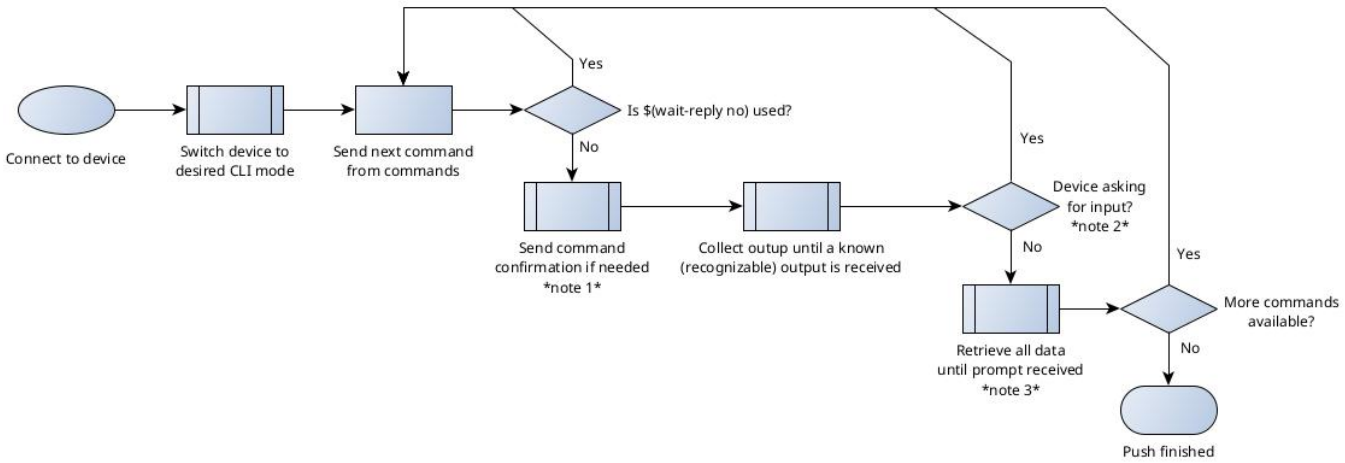
Please check the Command parsing examples section for more details on how modifiers and control sequences work.

## How commands are sent to devices

Config Push in Unimus has a few stages:

- connect to device, login to the CLI
- switch the device to desired CLI mode (if require "enable" or "configure" mode is set)
- send each command line from the "Commands" to the device (more info below)
- collect all output from device
- disconnect the CLI session
- create a new output group for this device OR assign the device into an existing output group

The main (and most important) portion of Config Push is sending the actual commands provided in the "Commands" set to the devices. To make this **simple** - you can provide the same sequence of commands and key-strokes to Unimus as you would when interacting with the device CLI, and Unimus will properly interact with the device as expected. To understand more **in depth** how Unimus interacts with a device, here is a (still very simplified) chart:




**Note 1:**
Unimus will automatically answer "Y" to questions like "Show all items?" or "Do you want to show sensitive items?", etc.

**Note 2:**
Unimus recognizes any generic questions asked by devices that end in the "[y/n]" or "(yes/no)" prompts, but also things like "Press enter to confirm change", etc.
Password prompts like "Password: " or "Password for..." are also properly recognized and will advance to sending the next line from the "Commands"

**Note 3:**
If the device pages through output ("-- more --"), Unimus sends pagination key (space for most device types) to proceed to next page automatically.
IF device asks to confirm end of output ("press Q to finish"), Unimus sends the appropriate key to finish output automatically.

# Command parsing examples

Push commands:

```
write memory
y
```

Device output (keys sent by Unimus in green):

```
device# write memory<enter>
This may take some time, continue? [y/n] y<enter>
..........
device#
```

For the "y" confirmation, it will be sent as "y" and then Enter. Here is an example of how to send a control sequence, and send the "y" without enter:

```
commit
$(send 0x1A)
y$(enter no)
```

Device output (keys sent by Unimus in green):

```
device# commit<enter>
Press CTRL + Z to continue, or Q to quit... <CTRL+Z>
Are you sure you want to contiue? [y/n] y
..... done
device#
```

Sometimes, you want Unimus to send commands to the device even if the device doesn't output any known outputs after sending a command line.
For example, many devices don't respond with anything if you need to provide a text block. You can avoid "INTERACTION_ERROR" in this case like this:

```
banner motd ^$(wait-reply no)
This is the start of the banner$(wait-reply no)
And this is a 2nd line of the banner$(wait-reply no)
This is the end of the banner^
```

It is important to NOT specify "$(wait-reply no)" after the last line, since the device will respond with a prompt after the last line, and you want Unimus to wait for the prompt and collect the output.

The interaction with the device will look like this:

```
device (config)# banner motd ^<enter>
Enter TEXT message.  End with the character '^'.
This is the start of the banner<enter>
And this is a 2nd line of the banner<enter>
This is the end of the banner^<enter>
device (config)#
```

# Debugging failed pushes - "INTERACTION_ERROR" group

If there is an "INTERACTION_ERROR" output group in your Push results, this means Unimus was not able to identify any known / expected output from some devices (see How commands are sent to devices). If this happens, Unimus will stop pushing to the device(s) for safety reasons, and place these devices in the "INTERACTION_ERROR" group.

The easiest way to find out why a push is failing is to enable Device Output Logging in "Zones > Your_zone > Debug Mode > Device output logging". After this is enabled, re-run your push and wait for it to fail.
After the push fails, download the Device Output Log file, and check what exactly in the device communication caused Unimus to fail.

You can also create a Support Ticket on our Portal and attach the log file - and we will be happy to investigate why the push failed.