Scripting

Scripting language manual

This manual provides an introduction to RouterOS's built-in powerful scripting language.

Scripting host provides a way to automate some router maintenance tasks by means of executing user-defined scripts bounded to some event occurrence.

Scripts can be stored in the Script repository or can be written directly to the console. The events used to trigger script execution include, but are not limited to the System Scheduler, the Traffic Monitoring Tool, and the Netwatch Tool generated events.

If you are already familiar with scripting in RouterOS, you might want to see our Tips & Tricks.

Line structure

The RouterOS script is divided into a number of command lines. Command lines are executed one by one until the end of the script or until a runtime error occurs.

Command-line

The RouterOS console uses the following command syntax:

`[prefix] [path] command [uparam] [param=[value]] .. [param=[value]]`

- [prefix] - ":" or "/" character which indicates if a command is ICE or path. It may not be required.
- [path] - relative path to the desired menu level. It may not be required.
- command - one of the commands available at the specified menu level.
- [uparam] - unnamed parameter, must be specified if the command requires it.
- [params] - a sequence of named parameters followed by respective values

The end of the command line is represented by the token *";"* or *NEWLINE*. Sometimes *";"* or *NEWLINE* is not required to end the command line.

Single command inside `(), [] or {}` does not require any end-of-command character. The end of the command is determined by the content of the whole script

```
:if ( true ) do={ :put "lala" }
```

Each command line inside another command line starts and ends with square brackets "[ ]" (command concatenation).

```
:put [/ip route get [find gateway=1.1.1.1]];
```

Notice that the code above contains three command lines:

- :put
- /ip route get
- find gateway=1.1.1.1

Command-line can be constructed from more than one physical line by following line joining rules.

Physical Line

A physical line is a sequence of characters terminated by an end-of-line (EOL) sequence. Any of the standard platform line termination sequences can be used:

- **Unix** – ASCII LF;
- **Windows** – ASCII CR LF;
- **mac** – ASCII CR;

Standard C conventions for newline characters can be used (the \n character).

Comments

The following rules apply to a comment:

- A comment starts with a hash character (#) and ends at the end of the physical line.
- RouterOS does not support multiline comments.
- If a **#** character appears inside the string it is not considered a comment.

Example

```
# this is a comment
# next line comment
:global a; # another valid comment

:global myStr "part of the string # is not a comment"
```

Line joining

Two or more physical lines may be joined into logical lines using the backslash character (\).

The following rules apply to using backslash as a line-joining tool:

- A line ending in a backslash cannot carry a comment.
- A backslash does not continue a comment.
- A backslash does not continue a token except for string literals.
- A backslash is illegal elsewhere on a line outside a string literal.

Example

```
:if ($a = true \
        and $b=false) do={ :put "$a $b"; }
:if ($a = true \ # bad comment
        and $b=false) do={ :put "$a $b"; }
# comment \
        continued - invalid (syntax error)
```

Whitespace between tokens

Whitespace can be used to separate tokens. Whitespace is necessary between two tokens only if their concatenation could be interpreted as a different token. Example:

```
{
        :local a true; :local b false;
# whitespace is not required
        :put (a&&b);
# whitespace is required
        :put (a and b);
}
```

Whitespace characters are not allowed

- between '<parameter>='
- between 'from=' 'to=' 'step=' 'in=' 'do=' 'else='

Example:

```
#incorrect:
:for i from = 1 to = 2 do = { :put $i }
#correct syntax:
:for i from=1 to=2 do={ :put $i }
:for i from= 1 to= 2 do={ :put $i }

#incorrect
/ip route add gateway = 3.3.3.3
#correct
/ip route add gateway=3.3.3.3
```

Scopes

Variables can be used only in certain regions of the script called scopes. These regions determine the visibility of the variable. There are two types of scopes - global and local. A variable declared within a block is accessible only within that block and blocks enclosed by it, and only after the point of declaration.

Global scope

Global scope or root scope is the default scope of the script. It is created automatically and can not be turned off.

⚠ Previously set global variable can be used in scripts by declaring it without setting the value.

Local scope

User can define their own groups to block access to certain variables, these scopes are called local scopes. Each local scope is enclosed in curly braces ("{ }").

```
{
        :local a 3;
        {
                :local b 4;
                :put ($a+$b);
        } #line below will show variable b in light red color since it is not defined in scope
        :put ($a+$b);
}
```

In the code above variable, b has local scope and will not be accessible after a closing curly brace.

Each line written in the terminal is treated as local scope

So for example, the defined local variable will not be visible in the next command line and will generate a syntax error

```
[admin@MikroTik] > :local myVar a;
[admin@MikroTik] > :put $myVar
syntax error (line 1 column 7)
```

Do not define global variables inside local scopes.

Note that even variable can be defined as global, it will be available only from its scope unless it is not referenced to be visible outside of the scope.

```
{
        :local a 3;
        {
                :global b 4;
        }
        :put ($a+$b);
}
```

The code above will output 3, because outside of the scope b is not visible.

The following code will fix the problem and will output 7:

```
{
        :local a 3;
        {
                :global b 4;
        }
        :global b;
        :put ($a+$b);
}
```

Keywords

The following words are keywords and cannot be used as variable and function names:

```
and       or       in
```

Delimiters

The following tokens serve as delimiters in the grammar:

```
()  []  {}  :   ;   $   /
```

Data types

RouterOS scripting language has the following data types:

Type	Description
num (number)	- 64bit signed integer, possible hexadecimal input;
bool (boolean)	- values can bee `true` or `false`;
str (string)	- character sequence;
ip	- IP address;
ip-prefix	- IP prefix;
ip6	- IPv6 address
ip6-prefix	- IPv6 prefix
id (internal ID)	- hexadecimal value prefixed by '*' sign. Each menu item has an assigned unique number - internal ID;
time	- date and time value;
array	- sequence of values organized in an array;
nil	- default variable type if no value is assigned;

Constant Escape Sequences

Following escape sequences can be used to define certain special characters within a string:

\"	Insert double quote
\	Insert backslash
\n	Insert newline
\r	Insert carriage return
\t	Insert horizontal tab
\$	Output $ character. Otherwise, $ is used to link the variable.
\?	~~Output ? character. Otherwise ? is used to print "help" in the console.~~ Removed since v7.1rc2
_	- space
\a	- BEL (0x07)
\b	- backspace (0x08)
\f	- form feed (0xFF)
\v	Insert vertical tab
\xx	A print character from hex value. Hex numbers should use capital letters.

Example

```
:put "\48\45\4C\4C\4F\r\nThis\r\nis\r\na\r\ntest";
```

which will show on the display

```
HELLO
This
is
a
test
```

Operators

Arithmetic Operators

Usual arithmetic operators are supported in the RouterOS scripting language

Operator	Description	Example
"+"	binary addition	`:put (3+4);`
"-"	binary subtraction	`:put (1-6);`
"*"	binary multiplication	`:put (4*5);`
"/"	binary division	`:put (10 / 2); :put ((10)/2)`
"%"	modulo operation	`:put (5 % 3);`
"-"	unary negation	`{ :local a 1; :put (-a); }`

Note: for the division to work you have to use braces or spaces around the dividend so it is not mistaken as an IP address

Relational Operators

Operator	Description	Example
"<"	less	`:put (3<4);`
">"	greater	`:put (3>4);`
"="	equal	`:put (2=2);`
"<="	less or equal	
">="	greater or equal	
"!="	not equal	

To negate an expression, you can use "expression=false". To print all interfaces that are not "ethernet", you can use expression negation like this:

```
/interface/print where (name~"ether")=false
```

Or to do the opposite, you can use "expression=true":

```
/interface/print where (name~"ether")=true
```

Logical Operators

Operator	Description	Example
“!”	logical NOT	`:put (!true);`
“&&”, “and”	logical AND	`:put (true&&true)`
“||”, “or”	logical OR	`:put (true||false);`
“in”		`:put (1.1.1.1/32 in 1.0.0.0/8);`

Bitwise Operators

Bitwise operators are working only on IP, and IPv6 address data types.

Operator	Description	Example

“~”	bit inversion	`:put (~0.0.0.0)` `:put (~::ffff)`
“|”	bitwise OR. Performs logical OR operation on each pair of corresponding bits. In each pair the result is “1” if one of the bits or both bits is “1”, otherwise the result is “0”.	`:put (192.168.88.0|0.0.0.255)` `:put (2001::1|::ffff)`
“^”	bitwise XOR. The same as OR, but the result in each position is “1” if two bits are not equal, and “0” if the bits are equal.	`:put (1.1.1.1^255.255.0.0)` `:put (2001::ffff:1^::ffff:0)`
“&”	bitwise AND. In each pair, the result is “1” if the first and second bit is “1”. Otherwise, the result is “0”.	`:put (192.168.88.77&255.255.255.0)` `:put (2001::1111&ffff::)`
“<<”	left shift by a given amount of bits, not supported for IPv6 address data type	`:put (192.168.88.77<<8)`
“>>”	right shift by a given amount of bits, not supported for IPv6 address data type	`:put (192.168.88.77>>24)`

Calculate the subnet address from the given IP and CIDR Netmask using the "&" operator:

```
{
:local IP 192.168.88.77;
:local CIDRnetmask 255.255.255.0;
:put ($IP&$CIDRnetmask);
}
```

Get the last 8 bits from the given IP addresses:

```
 :put (192.168.88.77&0.0.0.255);
```

Use the "|" operator and inverted CIDR mask to calculate the broadcast address:

```
{
:local IP 192.168.88.77;
:local Network 192.168.88.0;
:local CIDRnetmask 255.255.255.0;
:local InvertedCIDR (~$CIDRnetmask);
:put ($Network|$InvertedCIDR)
}
```

Concatenation Operators

Operator	Description	Example
"."	concatenates two strings	`:put ("concatenate" . " " . "string");`
","	concatenates two arrays or adds an element to the array	`:put ({1;2;3} , 5 );`

It is possible to add variable values to strings without a concatenation operator:

```
:global myVar "world";

:put ("Hello " . $myVar);
# next line does the same as above
:put "Hello $myVar";
```

By using $[] and $() in the string it is possible to add expressions inside strings:

```
:local a 5;
:local b 6;
:put " 5x6 = $($a * $b)";

:put " We have $[ :len [/ip route find] ] routes";
```

Other Operators

Operator	Description	Example
“[]”	command substitution. Can contain only a single command line	`:put [ :len "my test string"; ];`
“()”	subexpression or grouping operator	`:put ( "value is " . (4+5));`
“$”	substitution operator	`:global a 5; :put $a;`
“~”	the binary operator that matches value against POSIX extended regular expression	Print all routes whose gateway ends with 202: `/ip route print where gateway~"^[0-9 \\.]*202\$"`
“->”	Get an array element by key	`[admin@x86] >:global aaa {a=1;b=2}` `[admin@x86] > :put ($aaa->"a")` `1` `[admin@x86] > :put ($aaa->"b")` `2`

Variables

The scripting language has two types of variables:

- global - accessible from all scripts created by the current user, defined by global keyword;
- local - accessible only within the current scope, defined by local keyword.

There can be **undefined** variables. When a variable is undefined, the parser will try to look for variables set, for example, by DHCP lease-script or Hotspot on-login

Every variable, except for built-in RouterOS variables, must be declared before usage by local or global keywords. Undefined variables will be marked as undefined and will result in a compilation error. Example:

```
# following code will result in compilation error, because myVar is used without declaration
:set myVar "my value";
:put $myVar
```

Correct code:

```
:local myVar;
:set myVar "my value";
:put $myVar;
```

The exception is when using variables set, for example, by DHCP lease-script

```
/system script
add name=myLeaseScript policy=\
        ftp,reboot,read,write,policy,test,winbox,password,sniff,sensitive,api \
        source=":log info \$leaseActIP\r\
        \n:log info \$leaseActMAC\r\
        \n:log info \$leaseServerName\r\
        \n:log info \$leaseBound"

/ip dhcp-server set myServer lease-script=myLeaseScript
```

Valid characters in variable names are letters and digits. If the variable name contains any other character, then the variable name should be put in double quotes. Example:

```
#valid variable name
:local myVar;
#invalid variable name
:local my-var;
#valid because double quoted
:global "my-var";
```

If a variable is initially defined without value then the variable data type is set to *nil*, otherwise, a data type is determined automatically by the scripting engine. Sometimes conversion from one data type to another is required. It can be achieved using data conversion commands. Example:

```
#convert string to array
:local myStr "1,2,3,4,5";
:put [:typeof $myStr];
:local myArr [:toarray $myStr];
:put [:typeof $myArr]
```

Variable names are case-sensitive.

```
:local myVar "hello"
# following line will generate error, because variable myVAr is not defined
:put $myVAr
# correct code
:put $myVar
```

Set command without value will un-define the variable (remove from environment, new in v6.2)

```
#remove variable from environment
:global myVar "myValue"
:set myVar;
```

Use quotes on the full variable name when the name of the variable contains operators. Example:

```
:local "my-Var";
:set "my-Var" "my value";
:put $"my-Var";
```

Reserved variable names

All built-in RouterOS properties are reserved variables. Variables that will be defined the same as the RouterOS built-in properties can cause errors. To avoid such errors, use custom designations.

For example, the following script will not work:

```
{
:local type "ether1";
/interface print where name=$type;
}
```

But will work with different defined variables:

```
 {
:local customname "ether1";
/interface print where name=$customname;
}
```

Commands

Global commands

Every global command should start with the *":"* token, otherwise, it will be treated as a variable.

Command	Syntax	Description	Example
/		go to the root menu	
..		go back by one menu level	
?		list all available menu commands and brief descriptions	
global	`:global <var> [<value>]`	define a global variable	`:global myVar "something"; :put $myVar;`
local	`:local <var> [<value>]`	define the local variable	`{ :local myLocalVar "I am local"; :put $myVar; }`
beep	`:beep frequency=[num] length=[num]`	beep built-in speaker	

convert	`:convert from=[arg] to=[arg]`	Converts specified value from one format to another. By default uses an automatically parsed value, if the "from" format is not specified (for example, "001" becomes "1", "10.1" becomes "10.0.0.1", etc.).
		
		from specifies the format of the value - *base32, base64, byte-array, hex, num, raw, url.*
		
		to specifies the format of the output value - *base32, base64, bit-array-lsb, bit-array-msb, byte-array, hex, num, raw, url.*
		
		transform to transform values - *lc (transforms value to be in lowercases), uc (uppercases), lcfirst (first value to lowercase), ucfirst (first value to uppercase), crlf, ed25519-private-to-x25519-private, none, rot 13, x25519-private-to-x25519-public, ed25519-private-to-ed25519-public, ed25519-public-to-x25519-public, md5, reverse (reverses text), sha512.*
			`:put [:convert 001 to=hex ]`
			
			`31`
			
			`:put [:convert [/ip dhcp-client /option/get hostname raw-value] from=hex to=raw ]`
			
			`MikroTik`
			
			`:put [convert transform=lc "AAA"]`
			
			`aaa`

delay	`:delay <time>`	do nothing for a given period of time	
environment	`:environment print <start>`	print initialized variable information	`:global myVar true; :environment print;`
error	`:error <output>`	Generate console error and stop executing the script	

execute	`:execute <expression>`	Execute the script in the background. The result can be written in the file by setting a "file" parameter or printed to the CLI by setting "as-string".
		
		When using the "as-string" parameter executed script is blocked (not executed in the background).
		
		Executed script can not be larger than 64kB
			`{`
			`:local j [:execute {/interface print follow where [:log info ~Sname~]}];`
			`:delay 10s;`
			`:onerror e {/system script job remove $j}`
			`}`

find	`:find <arg> <arg> <start>`	return position of a substring or array element	`:put [:find "abc" "a" -1];`
grep	`:grep script=[str] pattern=[expression] after=[num] before=[num] filename=[str]`	Command ":grep" executes "script" part of a command in terminal and provides just the output which matches "pattern". Additional options "after" and "before" allow to print out specific number of lines of script output which are available also after/before matched pattern. Results can be directly saved into a file with "filename" parameter.	`:grep script="/interface print" pattern="ether" after=1 before=1 filename=results.txt`

jobname	:jobname	return current script name	**Limit script execution to single instance**
			`:if ([/system script job print count-only as-value where script=[:jobname] ] > 1) do={`
			`  :error "script instance already running"`
			`  }`

len	`:len <expression>`	return string length or array element count	`:put [:len "length=8"];`
log	`:log <topic> <message>`	write a message to the system log. Available topics are `"debug, error, info and warning"`	`:log info "Hello from script";`
onerror	`:onerror <var_name> in={<command>} do={<expression>}`	The command used to catch errors and get error details. The **do={...}** block is executed, when **in={...}** block has an error, and error details are written in <var_name> variable. Parameter order is important. The "error" parameter must be set before "do" block, otherwise do block will not see the local variable. :onerror will return *false* (if there is no error) and *true* (if there is an error) unless otherwise specified (with commands such as :return or :error), so it can be used in **:if** condition statement scripts.	`:onerror errorName in={ :error "failure" } do={ :put "Critical $errorName" }`

parse	`:parse <expression>`	parse the string and return parsed console commands. Can be used as a function.	`:global myFunc [:parse ":put hello!"];`
			`$myFunc;`

pick	`:pick <var> <start> [<end>]`	return range of elements or substring. If the count is not specified, will return only one element from an array.
		• var - value to pick elements from
		• start - element to start picking from (the first element index is 0)
		• end - terminating index (element at this index is not included)
			`[admin@MikroTik] > :put [:pick "abcde" 1 3]`
			`bc`

put	`:put <expression>`	put the supplied argument into the console	:put "Hello world"

range	`:range <var> <var>`	creates an array from the specified range	`:put [:range 2 8]`
			`2;3;4;5;6;7;8`

resolve	`:resolve <arg> [<domain-name>] [<server>] [<server-port>] [<type>]`	return the IP address of the given DNS name
		• domain-name - DNS name that needs to be resolved;
		• server - specific server that will be used to resolve DNS name (returned results will not be cached);
		• server-port - server port;
		• type - any/any6/ipv4/ipv6:
		  ○ any - first tries to resolve ipv4, if fails tries ipv6;
		  ○ any6 - first tries to resolve ipv6, if fails tries ipv4;
		  ○ ipv4 - tries to resolve only ipv4;
		  ○ ipv6 - tries to resolve only ipv6
			`:put [:resolve "www.mikrotik.com"];`
			
			`:put [:resolve domain-name="www.mikrotik.com"];`
			
			`:put [:resolve domain-name="www.mikrotik.com" server=192.168.88.1 port=53];`
			
			`:put [:resolve domain-name="www.mikrotik.com" type=ipv6];`

retry	:onerror e {:retry command=<expr> delay=[num] max=[num]} do={<expr>}	Try to execute the given command "max" amount of times with a given "delay" in seconds between tries. On failure, execute the command in the do={} block.	`:onerror e {:retry command={abc} delay=1 max=2} do={:put "got error"}`
			`got error`

typeof	`:typeof <var>`	the return data type of variable	`:put [:typeof 4];`

rndnum	`:rndnum from=[num] to=[num]`	random number generator	`:put [:rndnum from=1 to=99];`
rndstr	`:rndstr from=[str] length=[num]`	Random string generator. **from** specifies characters to construct the string from and defaults to all ASCII letters and numerals. **length** specifies the length of the string to create and defaults to 16.	`:put [:rndstr from="abcdef%^&" length=33];`
set	`:set <var> [<value>]`	assign value to a declared variable.	`:global a; :set a true;`

serialize	`:serialize [<value>] to=[arg]`	Serialize specified value/array to JSON or dsv (delimiter separated values) format.
		
		value specifues which values to process.
		
		to specifies the format - *json, dsv*
		
		delimiter sets the "separator".
		
		order specifies the order for variables.
		
		options specifies additional options*:*
		
		• *json.pretty* - makes the JSON output more visually appealing*;*
		• *json.no-string-conversion* - prevents implict conversions from console string type to json number type;
		• *dsv.wrap-strings* - wraps string values inside quatation marks;
		• *dsv.ignore-size* - if array values have different sizes, e.g. `a=(1,2);b=(3,4);c=(5,6,7)`, this option will work around `array size mismatch` error and set "empty" values in those slots.
		• *dsv.remap* - merges array of dictionaries into a single dictionary (useful when working with "`print as-value`")
		
		file-name enables the option to generate command's output into a file (available for download in the "/files" section).
			`:put [:serialize value=a,b,c`
			`to=json]`
			`["a","b","c"]`
			
			`:local test {a=(1,2,3);b=(4,5,6);c=(7,"text",9)}; :put [ :serialize to=dsv delimiter=";" value=$test order=("c","a","b") ]`
			`c;a;b`
			`7;1;4`
			`text;2;5`
			`9;3;6`
			
			`:global var ({ "string"="1234"; "number"=1234 });:put [ :serialize to=json value=$var ]`
			`{"number":1234,"string":1234.000000}`
			`:put [ :serialize to=json value=$var options=json.no-string-conversion  ]`
			`{"number":1234,"string":"1234"}`
			
			`:put [:serialize to=dsv options=dsv.remap delimiter="#" [/ip/address/print as-value]]`
			`.id#address#comment#interface#network`
			`*1#192.168.88.1/24#defconf#bridge#192.168.88.0`
			`*2#192.168.69.190/24##ether1#192.168.69.0`

deserialize	`:deserialize [<value>] from=[arg]`	Deserialize specified value/array from JSON or dsv (delimiter separated values) format.
		
		from specifies the format - *json, dsv*
		
		delimiter sets the "separator".
		
		options specifies additional options*:*
		• *dsv.plain* - deserializes every line as an array (input does not have a header or column names);
		• *dsv.array* - expects a header (column names) and will return an array of dictionaries, where values are mapped to column names provided in the header.
		• *json.no-string-conversion* - prevents implicit conversions from json string type to console values (number, ip, etc.).
			`:put [:deserialize from=json value="[\"a\",\"b\",\"c\"]"]`
			`a;b;c`
			
			`:put ([ :deserialize from=dsv delimiter=";" value="a;b;c\n1;findme;3" options=dsv.plain ]->1->1)`
			`findme`
			
			`:put ([ :deserialize from=dsv delimiter=";" value="a;b;c\n1;findme;3" options=dsv.plain ]->0->1)`
			`b`
			
			`:put ([:deserialize from=dsv "a;b;c\n1;2;3\n4;5;6" delimiter=";" options=dsv.array]->1->"b")`
			`5`
			
			`:put ([:deserialize from=dsv "a;b;c\n1;2;3\n4;5;6" delimiter=";" options=dsv.array]->0->"c")`
			`3`
			
			`:put [typeof ([:deserialize "{ \"str\": \"123\" }" from=json options=json.no-string-conversion]->"str")]`
			`str`
			
			`:deserialize [/file/get file.json contents] from=json`

time	`:time <expression>`	return interval of time needed to execute the command	`:put [:time {:for i from=1 to=10 do={ :delay 100ms }}];`

timestamp	`:timestamp`	returns the time since epoch, where epoch is January 1, 1970 (Thursday), not counting leap seconds	`[admin@MikroTik] > :put [:timestamp]`
			`2735w21:41:43.481891543`
			`or`
			`[admin@MikroTik] > :put [:timestamp]`
			`2735w1d21:41:43.481891543`
			`with the day offset`

toarray	`:toarray <var>`	convert a variable to the array	
tobool	`:tobool <var>`	convert a variable to boolean	
toid	`:toid <var>`	convert a variable to internal ID	
toip	`:toip <var>`	convert a variable to IP address	
toip6	`:toip6 <var>`	convert a variable to IPv6 address	
tonum	`:tonum <var>`	convert a variable to an integer	

tostr	`:tostr <var>`	convert a variable to a string	
totime	`:totime <var>`	convert a variable to time	

tonsec	`:tonsec <var>`	convert time to nanoseconds	`:put [:tonsec value=10:`
			`00]`
			`36000000000000`

tocrlf	`:tocrlf <var>`	converts line endings to CRLF	`:put [:tocrlf  "AAA\r\nBBB\r\nCC`
			`C"`
			`]`
			
			`AAA`
			
			`BBB`
			`CCC`

tolf	`:tolf <var>`	converts line endings to LF	`:put [:tolf  "AAA\nBBB\nCCC"`
			`]`
			`AAA`
			`   BBB`
			`      CCC`

nothing	`:nothing`	return a value of nothing	`:if ([:nothing] = 0) do={:put`
			`true} else={:put false}`
			`false`
			`:if ([:nothing] > 0) do={:put`
			`true} else={:put false}`
			`false`
			`:if ([:nothing] < 0) do={:put`
			`true} else={:put false}`
			`true`

If a variable type conversion function cannot apply the new format to the provided data, the output will be empty.

For example, if you run the :tonum <var> command on a variable with a non-integer value such as "23.8" or "cow&chicken", the result will be empty, and its data type will be shown as nil.

Menu specific commands

Common commands

The following commands are available from most sub-menus:

Command	Syntax	Description
add	`add <param>=<value>..<param>=<value>`	add new item
remove	`remove <id>`	remove selected item
enable	`enable <id>`	enable selected item
disable	`disable <id>`	disable selected item

set	`set <id> <param>=<value>..<param>=<value>`	change selected items parameter, more than one parameter can be specified at the time. The parameter can be unset by specifying '!' before the parameter.
		
		Example:
		`/ip firewall filter add chain=blah action=accept protocol=tcp port=123 nth=4,2`
		`print`
		`set 0 !port chain=blah2 !nth protocol=udp`

get	`get <id> <param>=<value>`	get the selected item's parameter value

print	`print <param><param>=[<value>]`	print menu items. Output depends on the print parameters specified. The most common print parameters are described here
export	`export [file=<value>]`	export configuration from the current menu and its sub-menus (if present). If the file parameter is specified output will be written to the file with the extension '.rsc', otherwise the output will be printed to the console. Exported commands can be imported by import command
edit	`edit <id> <param>`	edit selected items property in the built-in text editor
find	`find <expression>`	Returns list of internal numbers for items that are matched by given expression. For example: `:put [ /interface find name~"ether"]`

import

The import command is available from the root menu and is used to import configuration from files created by an export command or written manually by hand.

Starting from 7.16.x version, its possible to catch syntax errors:

```
[admin@admin] > do { import test.rsc } on-error={ :put "Failure" }
Failure
```

New parameter *onerror* can be used:

```
[admin@admin] > onerror e in={ import test.rsc } do={ :put "Failure - $e" }
Failure - Script Error: bad command name this (line 1 column 1)
```

In addition, the *import* command has new options in *verbose* mode - the *dry-run* parameter is specially designed for debugging and can find multiple errors without changing the configuration.

```
[admin@admin] > import test.rsc verbose=yes dry-run
#line 1
this
bad command name this (line 1 column 1)
...
Script Error: found 5 error(s) in import file
```

print parameters

Several parameters are available for print command:

Parameter	Description	Example
append		
as-value	print output as an array of parameters and its values	`:put [/ip address print as-value]`
brief	print brief description	
detail	print detailed description, the output is not as readable as brief output but may be useful to view all parameters	
count-only	print only count of menu items	
file	print output to a file	
follow	print all current entries and track new entries until ctrl-c is pressed, very useful when viewing log entries	`/log print follow`
follow-only	print and track only new entries until ctrl-c is pressed, very useful when viewing log entries	`/log print follow-only`

from	print parameters only from specified item	`/user print from=admin`
interval	continuously print output in a selected time interval, useful to track down changes where `follow` is not acceptable	`/interface print interval=2`
terse	show details in a compact and machine-friendly format	
value-list	show values single per line (good for parsing purposes)	
without-paging	If the output does not fit in the console screen then do not stop, print all information in one piece	
where	expressions followed by where parameters can be used to filter outmatched entries	`/ip route print where interface="ether1"`
about	returns entries that have the "about" parameter, such as "managed by CAPsMAN "information or warnings	`/interface wifi print where about`

More than one parameter can be specified at a time, for example, `/ip route print count-only interval=1 where interface="ether1"`

Loops and conditional statements

Loops

Command	Syntax	Description
do..while	`:do { <commands> } while=( <conditions> ); :while ( <conditions> ) do={ <commands> };`	execute commands until a given condition is met.
for	`:for <var> from=<int> to=<int> step=<int> do={ <commands> }`	execute commands over a given number of iterations
foreach	`:foreach <var> in=<array> do={ <commands> };`	execute commands for each element in a list

Conditional statement

Command	Syntax	Description
if	`:if (<condition>) do={<commands>} else={<commands>}`	If a given condition is `true` then execute commands in the `do` block, otherwise execute commands in the `else` block if specified.

Example:

```
{
        :local myBool true;
        :if ($myBool = false) do={ :put "value is false" } else={ :put "value is true" }
}
```

Functions

Scripting language does not allow you to create functions directly, however, you could use :parse command as a workaround.

Starting from v6.2 new syntax is added to easier define such functions and even pass parameters. It is also possible to return function value with **:return** co mmand.

See examples below:

```
#define function and run it
:global myFunc do={:put "hello from function"}
$myFunc

output:
hello from function

#pass arguments to the function
:global myFunc do={:put "arg a=$a"; :put "arg '1'=$1"}
$myFunc a="this is arg a value" "this is arg1 value"

output:
arg a=this is arg a value
arg '1'=this is arg1 value
```

Notice that there are two ways how to pass arguments:

- pass arg with a specific name ("a" in our example)
- pass value without arg name, in such case arg "1", "2" .. "n" is used.

Return example

```
:global myFunc do={ :return ($a + $b)}
:put [$myFunc a=6 b=2]

output:
8
```

You can even clone an existing script from the script environment and use it as a function.

```
#add script
/system script add name=myScript source=":put \"Hello \$myVar !\""

:global myFunc [:parse [/system script get myScript source]]
$myFunc myVar=world

output:
Hello world !
```

If the function contains a defined global variable that names match the name of the passed parameter, then the globally defined variable is ignored, for compatibility with scripts written for older versions. This feature can change in future versions. **Avoid using parameters with the same name as global variables.**

For example:

```
:global my2 "123"

:global myFunc do={ :global my2; :put $my2; :set my2 "lala"; :put $my2 }
$myFunc my2=1234
:put "global value $my2"
```

The output will be:

```
1234
lala
global value 123
```

Nested function example

Note: to call another function its name needs to be declared (the same as for variables)

```
:global funcA do={ :return 5 }
:global funcB do={
        :global funcA;
        :return ([$funcA] + 4)
}
:put [$funcB]

Output:
9
```

Catch run-time errors

Starting from v6.2 scripting has the ability to catch run-time errors.

For example, the [code]:reslove[/code] command if failed will throw an error and break the script.

```
[admin@MikroTik] > { :put [:resolve www.example.com]; :put "lala";}
failure: dns name does not exist
```

Now we want to catch this error and proceed with our script:

```
:onerror e {:put [:resolve www.example.com]} do={:put "resolver failed"}
:put "lalala"

output:

resolver failed
lala
```

Operations with Arrays

Warning: Key name in the array contains any character other than a lowercase character, it should be put in quotes

For example:

```
[admin@ce0] > {:local a { "aX"=1; ay=2 }; :put ($a->"aX")}
1
```

Loop through keys and values

"foreach" command can be used to loop through keys and elements:

```
[admin@ce0] > :foreach k,v in={2; "aX"=1; y=2; 5} do={:put ("$k=$v")}

0=2
1=5
aX=1
y=2
```

If the "foreach" command is used with one argument, then the element value will be returned:

```
[admin@ce0] > :foreach v in={2; "aX"=1; y=2; 5} do={:put ("$v")}

2
5
1
2
```

Note: If the array element has a key then these elements are sorted in alphabetical order, elements without keys are moved before elements with keys and their order is not changed (see example above).

Change the value of a single array element

```
[admin@MikroTik] > :global a {x=1; y=2}
[admin@MikroTik] > :set ($a->"x") 5
[admin@MikroTik] > :environment print
a={x=5; y=2}
```

Script repository

Sub-menu level: `/system script`

Contains all user-created scripts. Scripts can be executed in several different ways:

- **on event** - scripts are executed automatically on some facility events (scheduler, netwatch, VRRP)
- **by another script** - running script within the script is allowed
- **manually** - from console executing a run command or in winbox

Only scripts (including schedulers, netwatch, etc) with equal or higher permission rights can execute other scripts.

When executing script from GUI or CLI, user permissions are used. To run a script with script permissions, a script must be executed from CLI with additional "*use-script-permissions*" parameter.

Property	Description
comment (*string*; Default:)	Descriptive comment for the script
dont-require-permissions (*yes | no*; Default: *no*)	Bypass permissions check when the script is being executed, useful when scripts are being executed from services that have limited permissions, such as Netwatch
name (*string*; Default: *"Script[num]"*)	name of the script
policy (*string*; Default: ftp,reboot,read,write,policy, test,password,sniff,sensitive,romon)	list of applicable policies: • **ftp** - can log on remotely via FTP and send and retrieve files from the router • **password** - change passwords • **policy** - manage user policies, add and remove user • **read** - can retrieve the configuration • **reboot** - can reboot the router • **sensitive** - allows changing "hide sensitive" parameter • **sniff** - can run sniffer, torch, etc • **test** - can run ping, traceroute, bandwidth test • **write** - can change the configuration Read more detailed policy descriptions here
source (*string*;)	Script source code

Read-only status properties:

Property	Description
last-started (*date*)	Date and time when the script was last invoked.
owner (*string*)	The user who created the script
run-count (*integer*)	Counter that counts how many times the script has been executed

Menu specific commands

Command	Description
run (*run [id|name]*)	Execute the specified script by ID or name using user permissions.
use-script-permissions	Additional parameter to execute script using script permissions.

Environment

Sub-menu level:

- `/system script environment`
- `/environment`

Contains all user-defined variables and their assigned values.

```
[admin@MikroTik] > :global example;
[admin@MikroTik] > :set example 123
[admin@MikroTik] > /environment print
"example"=123
```

Read-only status properties:

Property	Description
name (*string*)	Variable name
user (*string*)	The user who defined variable
value ()	The value assigned to a variable

Job

Sub-menu level: `/system script job`

Contains a list of all currently running scripts.
Read-only status properties:

Property	Description
owner (*string*)	The user who is running the script
policy (*array*)	List of all policies applied to the script
started (*date*)	Local date and time when the script was started

Script permissions

There are four ways a script can be run: using script permissions, user permissions, scheduler permissions or on-event permissions (such as `/system routerboard mode-button` settings).

Depending on how a script is called, it may inherit different permissions or use its own.

When using `/system script run`, a script can inherit the permissions of the caller.

In this example, we use an admin user with full permissions and test running a script both with and without the `use-script-permissions` parameter.

```
/system script
add dont-require-permissions=no name=add-dhcp-no-perms owner=admin policy="" sour
ce="/ip dhcp-client add interface=ether2; put \"Added DHCP client ether2!\""

/user/print
Columns: NAME, GROUP, LAST-LOGGED-IN, INACTIVITY-POLICY
# NAME   GROUP  LAST-LOGGED-IN       INACTIVITY-POLICY
;;; system default user
0 admin  full   2025-07-22 17:09:59  none
```

```
[admin@MikroTik] > system/script/run add-dhcp-no-perms  use-script-permissions
not enough permissions (9)
[admin@MikroTik] > system/script/run add-dhcp-no-perms
Added DHCP client on ether2!
```

Similarly, there are multiple ways to run a script using the scheduler tool. When using the scheduler, the script can run with the scheduler's permissions.

You can also call the script by name using the scheduler, which works the same way as `/system script run use-script-permissions`.

To demonstrate this, we create three schedulers, each configured to run the script in a different way:

```
/system scheduler
add interval=10s name=run-script-use-script-perms on-event="/system script run add-dhcp-no-perms use-script-
permissions" policy=ftp,reboot,read,write,policy,test,password,sniff,sensitive,romon
add interval=10s name=run-script-direct on-event=add-dhcp-no-perms policy=ftp,reboot,read,write,policy,test,
password,sniff,sensitive,romon
add interval=10s name=run-script-scheduler-perms on-event="/system script run add-dhcp-no-perms" policy=ftp,
reboot,read,write,policy,test,password,sniff,sensitive,romon
```

After these schedulers run, the logs show that the two methods using `use-script-permissions` or calling the script by name fail due to insufficient permissions.

In contrast, `run-script-scheduler-perms` executes the script successfully, as it inherits the scheduler's permissions.

```
/log print
 2025-07-22 18:11:25 script,error executing script add-dhcp-no-perms from scheduler (run-script-direct) failed,
please check it manually
 2025-07-22 18:11:25 script,error,debug not enough permissions (9) (/ip/dhcp-client/add; line 1)
 2025-07-22 18:11:25 script,error executing script from scheduler (run-script-use-script-perms) failed, please
check it manually
 2025-07-22 18:11:25 script,error,debug (scheduler:run-script-use-script-perms) not enough permissions (9) (/ip
/dhcp-client/add; line 1)
 2025-07-22 18:11:25 system,info dhcp client added by scheduler:run-script-scheduler-perms/script:add-dhcp-no-
perms (*7 = /ip dhcp-client add interface=ether2)
```

A script with higher or more permissions than the user/scheduler cannot be run; use-script-permissions won’t override this.

See also

- Scripting Examples
- Manual: Scripting Tips and Tricks

Videos on Scripting

Scripting pt1

Scripting pt2

Scripting: Data Types

Scripting: Arrays