Manual Scripting Mikrotik RouterOS

https://wiki.mikrotik.com/wiki/Manual%3AScripting

Contents

Scripting language manual

This manual provides introduction to RouterOS 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 Script repository or can be written directly to 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.

Line structure

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

Command line

RouterOS console uses following command syntax:

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

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

The end of 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. End of command is determined by content of 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 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 new line characters can be used (the \n character).

Comments

A comment starts with a hash character (#) and ends at the end of the physical line. Whitespace or any other symbols are not allowed before hash symbol. Comments are ignored by syntax. If (#) character appear inside string it is not considered a comment.

Example

```
# this is a comment
 # bad comment
:global a; # bad comment
```

```
:global myStr "lala # this is not a comment"
```

Line joining

Two or more physical lines may be joined into logical lines using backslash character (\). 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 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. These regions are called scopes. Scope determines 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 default scope of the script. It is created automatically and can not be turned off.

Local scope

User can define its 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 code above variable b has local scope and will not be accessible after closed curly brace.

Note: Each line written in terminal is treated as local scope

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

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

Warning: 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 already defined.

```
{
   :local a 3;
   {
       :global b 4;
```

```
    }
    :put ($a+$b);
}
```

Code above will generate an error.

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 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-prefix	- IPv6 prefix
id (internal ID)	- hexadecimal value prefixed by '*' sign. Each menu item has 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 character within string:

\"	Insert double quote
****	Insert backslash
\n	Insert newline
\r	Insert carriage return
\t	Insert horizontal tab
\$	Output $ character. Otherwise $ is used to link variable.
\?	Output ? character. Otherwise ? is used to print "help" in console.
_	- space
\a	- BEL (0x07)
\b	- backspace (0x08)

\f	- form feed (0xFF)
\v	Insert vertical tab
\xx	Print character from hex value. Hex number should use capital letters.

Example

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

which will show on display

```
HELLO
This
is
a
test
```

Operators

Arithmetic Operators

Usual arithmetic operators are supported in RouterOS scripting language

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

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

Relational Operators

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

Logical Operators

Opearator	Description	Exampl
"!"	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 on number and ip address data types.

Opearator	Description
"~"	bit inversion
"|"	bitwise OR. Performs logical OR operation on each pair of corresponding bits. In each pair the result is "1" if one of both bits are "1", otherwise the result is "0".
"^"	bitwise XOR. The same as OR, but the result in each position is "1" if two bits are not equal, and "0" if bits are equa
"&"	bitwise AND. In each pair the result is "1" if first and second bit is "1". Otherwise the result is "0".
"<<"	left shift by given amount of bits
">>"	right shift by given amount of bits

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

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

Get last 8 bits from given IP addresses:

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

Use "|" 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

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

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

```
:global myVar "world";
```

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

By using $[] and $() in 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

</tr>

Opearator	Description	
"[]"	command substitution. Can contain only single command line	`:put [ :len "my test str`
"()"	sub expression or grouping operator	`:put ( "value is " . (4-`
"$"	substitution operator	`:global a 5; :put $a;`
"~"	binary operator that matches value against POSIX extended regular expression	Print all routes which gateway enc `/ip route print where ga`
"->"	Get an array element by key	`[admin@x86] >:global a` `[admin@x86] > :put ($a` `1` `[admin@x86] > :put ($a` `2`

Variables

Scripting language has two types of variables:

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

Note: Starting from v6.2 there can be undefined variables. When variable is undefined parser will try to look for variables set, for example, by DHCP lease-script or Hotspoton-login

Note: Variable value size is limited to 4096bytes

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 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;
```

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 variable name contains any other character, then 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 variable is initially defined without value then variable data type is set to *nil*, otherwise data type is determined automatically by 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;
```

Commands

Global commands

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

Command	Syntax	Description
/		go to root menu
..		go back by one menu level
?		list all available menu commands and brief descriptions
global	`:global <var> [<value>]`	define global variable
local	`:local <var> [<value>]`	define local variable
beep	`:beep <freq> <length>`	beep built in speaker
delay	`:delay <time>`	do nothing for a given period of time
put	`:put <expression>`	put supplied argument to console
len	`:len <expression>`	return string length or array element count
typeof	`:typeof <var>`	return data type of variable
pick	`:pick <var> <start>[<end>]`	return range of elements or substring. If end position is not specified, will ret from an array.
log	`:log <topic> <message>`	write message to system log. Available topics are `"debug, error, info`
time	`:time <expression>`	return interval of time needed to execute command
set	`:set <var> [<value>]`	assign value to declared variable.
find	`:find <arg> <arg> <start>`	return position of substring or array element
environment	`:environment print <start>`	print initialized variable information

terminal		terminal related commands
error	`:error <output>`	Generate console error and stop executing the script
execute	`:execute <expression>`	Execute the script in background.
parse	`:parse <expression>`	parse string and return parsed console commands. Can be used as function.
resolve	`:resolve <arg>`	return IP address of given DNS name
toarray	`:toarray <var>`	convert variable to array
tobool	`:tobool <var>`	convert variable to boolean
toid	`:toid <var>`	convert variable to internal ID
toip	`:toip <var>`	convert variable to IP address
toip6	`:toip6 <var>`	convert variable to IPv6 address
tonum	`:tonum <var>`	convert variable to integer
tostr	`:tostr <var>`	convert variable to string
totime	`:totime <var>`	convert variable to time

Menu specific commands

Common commands

Following commands available from most sub-menus:

Command	Syntax	Descript
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 by specifying '!' before parameter. Example: `/ip firewall filter add chain=blah action=`

		`print` `set 0 !port chain=blah2 !nth protocol=udp`
get	`get <id> <param>=<value>`	get selected items parameter value
print	`print <param><param>=[<value>]`	print menu items. Output depends on print parameters speci described here
export	`export [file=<value>]`	export configuration from current menu and its sub-menus be written to file with extension '.rsc', otherwise output will imported by import command
edit	`edit <id> <param>`	edit selected items property in built-in text editor
find	`find <expression>`	Returns list of internal numbers for items that are matched `[/interface find name~"ether"]`

import

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

print parameters

Several parameters are available for print command:

Parameter	Description
append	
as-value	print output as array of parameters and its values
brief	print brief description
detail	print detailed description, 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 file
follow	print all current entries and track new entries until ctrl-c is pressed, very useful when viewing log entries
follow-only	print and track only new entries until ctrl-c is pressed, very useful when viewing log entries
from	print parameters only from specified item
interval	continuously print output in selected time interval, useful to track down changes where `follow` is not acceptable
terse	show details in compact and machine friendly format
value-list	show values one per line (good for parsing purposes)
without-paging	If output do not fit in console screen then do not stop, print all information in one piece
where	expressions followed by where parameter can be used to filter out matched entries

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
do..while	`:do { <commands> } while=( <conditions> ); :while ( <conditions> ) do={ <commands> }`
for	`:for <var> from=<int> to=<int> step=<int> do={ <commands> }`

foreach	`:foreach <var> in=<array> do={ <commands> };`

Conditional statement

Command	Syntax	
if	`:if(<condition>) do={<commands>} else={<commands>} <expression>`	If a given condition is `true` execute commands in the `e`

Example:

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

Functions

Scripting language does not allow 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** command.

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

Return example

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

output:
8
```

You can even clone existing script from script environment and use it as 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 !
```

> **Warning:** If function contains defined global variable which name matches the name of passed parameter, then globally defined variable is ignored, for compatibility with scripts written for older versions. This feature can change in future versions. **Avoid using parameters with 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"
```

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 ability to catch run-time errors.

For example, [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:

```
:do {
      :put [:resolve www.example.com];
} on-error={ :put "resolver failed"};
:put "lala"

output:

resolver failed
lala
```

Operations with Arrays

Warning: Key name in array contains any character other than 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 foreach command is used with one argument, then element value will be returned:

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

2
5
1
2
```

Note: If array element has 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 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 script is allowed
- **manually** - from console executing run command or in winbox

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

Property	
comment (*string*; Default:)	Descriptive comment for the script
dont-require-permissions (*yes* | *no*; Default: *no*)	Bypass permissions check when script is being that have limited permissions, such as Netwatch
name (*string*; Default: *"Script[num]"*)	name of the script
policy (*string*; Default:)	list of applicable policies: • **api** - api permissions • **ftp** - can log on remotely via ftp and se

- **local** - can log on locally via console
- **password** - change passwords
- **policy** - manage user policies, add and
- **read** - can retrieve the configuration
- **reboot** - can reboot the router
- **sensitive** - allows to change "hide sensi
- **sniff** - can run sniffer, torch etc
- **ssh** - can log on remotely via secure sh
- **telnet** - can log on remotely via telnet
- **test** - can run ping, traceroute, bandwid
- **web** - can log on remotely via http
- **winbox** - winbox permissions
- **write** - can change the configuration

Read more detailed policy descriptions here

source (*string*;)	Script source code

Read only status properties:

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

Menu specific commands

Command	
run (*run [id|name]*)	Execute specified script by ID or name

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	
name (*string*)	Variable name
user (*string*)	User who defined variable
value ()	Value assigned to variable

Job

Sub-menu level: `/system script job`

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

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

Manual:Scripting-examples

Contents

[hide]

CMD Scripting examples

This section contains some useful scripts and shows all available scripting features. Script examples used in this section were tested with the latest 3.x version.

Create a file

In v3.x it is not possible to create file directly, however there is a workaround

```
/file print file=myFile
/file set myFile.txt contents=""
```

Check if IP on interface have changed

Sometimes provider gives dynamic IP addresses. This script will compare if dynamic IP address is changed.

```
:global currentIP;

:local newIP [/ip address get [find interface="ether1"] address];
```

```
:if ($newIP != $currentIP) do={
    :put "ip address $currentIP changed to $newIP";
    :set currentIP $newIP;
}
```

Strip netmask

This script is useful if you need ip address without netmask (for example to use it in firewall), but "`/ip address get [id] address`" returns ip address and netmask.

Code:

```
:global ipaddress 10.1.101.1/24

:for i from=( [:len $ipaddress] - 1) to=0 do={
     :if ( [:pick $ipaddress $i] = "/") do={
             :put [:pick $ipaddress 0 $i]
     }
}
```

Another much more simple way:

```
:global ipaddress 10.1.101.1/24
:put [:pick $ipaddress 0 [:find $ipaddress "/"]]
```

Resolve host-name

Many users are asking feature to use dns names instead of IP address for radius servers, firewall rules, etc.

So here is an example how to resolve RADIUS server's IP.

Lets say we have radius server configured:

```
/radius
add address=3.4.5.6 comment=myRad
```

And here is a script that will resolve ip address, compare resolved ip with configured one and replace if not equal:

```
/system script add name="resolver" source= {

:local resolvedIP [:resolve "server.example.com"];
:local radiusID [/radius find comment="myRad"];
:local currentIP [/radius get $radiusID address];

:if ($resolvedIP != $currentIP) do={
   /radius set $radiusID address=$resolvedIP;
   /log info "radius ip updated";
}
```

```
}
```

Add this script to scheduler to run for example every 5 minutes

```
/system scheduler add name=resolveRadiusIP on-event="resolver" interval=5m
```

Write simple queue stats in multiple files

Lets consider queue namings are "some text.1" so we can search queues by last number right after the dot.

```
:local entriesPerFile 10;
:local currentQueue 0;
:local queuesInFile 0;
:local fileContent "";
#determine needed file count
:local numQueues [/queue simple print count-only] ;
:local fileCount ($numQueues / $entriesPerFile);
:if ( ($fileCount * $entriesPerFile) != $numQueues) do={
   :set fileCount ($fileCount + 1);
}

#remove old files
/file remove [find name~"stats"];

:put "fileCount=$fileCount";

:for i from=1 to=$fileCount do={
#create file
   /file print file="stats$i.txt";
#clear content
   /file set [find name="stats$i.txt"] contents="";

   :while ($queuesInFile < $entriesPerFile) do={
     :if ($currentQueue < $numQueues) do={
         :set currentQueue ($currentQueue +1);
         :put $currentQueue ;
         /queue simple
         :local internalID [find name~"\\.$currentQueue\$"];
         :put "internalID=$internalID";
         :set fileContent ($fileContent . [get $internalID target-address] . \
           " " . [get $internalID total-bytes] . "\r\n");
     }
     :set queuesInFile ($queuesInFile +1);

   }
   /file set "stats$i.txt" contents=$fileContent;
   :set fileContent "";
```

```
   :set queuesInFile 0;

}
```

Generate backup and send it by e-mail

This script generates backup file and sends it to specified e-mail address. Mail subject contains router's name, current date and time.

Note that smtp server must be configured before this script can be used. See `/tool e-mail` for configuration options.

Script:

```
/system backup save name=email_backup
/tool e-mail send file=email_backup.backup to="me@test.com" body="See attached file" \
   subject="$[/system identity get name] $[/system clock get time] $[/system clock get
date] Backup")
```

> **Note:** backup file contains sensitive information like passwords. So to get access to generated backup file, script or scheduler must have '**sensitive**' policy.

Use string as function

Code:

```
:global printA [:parse ":local A; :put \$A;" ];
$printA
```

Check bandwidth and add limitations

This script checks if download on interface is more than 512kbps, if true then queue is added to limit speed to 256kbps.

Code:

```
:foreach i in=[/interface find] do={
    /interface monitor-traffic $i once do={
        :if ($"received-bits-per-second" > 0 ) do={
            :local tmpIP [/ip address get [/ip address find interface=$i] address] ;
#            :log warning $tmpIP ;
            :for j from=( [:len $tmpIP] - 1) to=0 do={
                :if ( [:pick $tmpIP $j] = "/") do={
                    /queue simple add name=$i max-limit=256000/256000 dst-
address=[:pick $tmpIP 0 $j] ;
                }
            }
```

```
        }
    }
}
```

Block access to specific websites

This script is useful if you want to block certain web sites but you don't want to use web proxy.

This example looks entries "rapidshare" and "youtube" in dns cache and adds IPs to address list named "restricted".

Before you begin, you must set up router to catch all dns requests:

```
/ip firewall nat
add action=redirect chain=dstnat comment=DNS dst-port=53 protocol=tcp to-ports=53
add action=redirect chain=dstnat dst-port=53 protocol=udp to-ports=53
```

and add firewall

```
/ip firewall filter
add chain=forward dst-address-list=restricted action=drop
```

Now we can write a script and schedule it to run, lets say, every 30 seconds.

Script Code:

```
:foreach i in=[/ip dns cache find] do={
    :local bNew "true";
    :local cacheName [/ip dns cache all get $i name] ;
#    :put $cacheName;

    :if (([:find $cacheName "rapidshare"] >= 0) || ([:find $cacheName "youtube"] >= 0))
do={

        :local tmpAddress [/ip dns cache get $i address] ;
#    :put $tmpAddress;

# if address list is empty do not check
        :if ( [/ip firewall address-list find list="restricted" ] = "") do={
            :log info ("added entry: $[/ip dns cache get $i name] IP $tmpAddress");
            /ip firewall address-list add address=$tmpAddress list=restricted
comment=$cacheName;
        } else={
            :foreach j in=[/ip firewall address-list find list="restricted"] do={
                :if ( [/ip firewall address-list get $j address] = $tmpAddress ) do={
                    :set bNew "false";
                }
            }
            :if ( $bNew = "true" ) do={
                :log info ("added entry: $[/ip dns cache get $i name] IP $tmpAddress");
```

```
                /ip firewall address-list add address=$tmpAddress list=restricted
comment=$cacheName;
            }
        }
    }
}
```

Parse file to add ppp secrets

This script requires that entries inside the file is in following format:

username,password,local_address,remote_address,profile,service

For example:

```
janis,123,1.1.1.1,2.2.2.1,ppp_profile,myService
juris,456,1.1.1.1,2.2.2.2,ppp_profile,myService
aija,678,1.1.1.1,2.2.2.3,ppp_profile,myService
```

Code:

```
:global content [/file get [/file find name=test.txt] contents] ;
:global contentLen [ :len $content ] ;

:global lineEnd 0;
:global line "";
:global lastEnd 0;


:do {
       :set lineEnd [:find $content "\r\n" $lastEnd ] ;
       :set line [:pick $content $lastEnd $lineEnd] ;
       :set lastEnd ( $lineEnd + 2 ) ;

       :local tmpArray [:toarray $line] ;
     :if ( [:pick $tmpArray 0] != "" ) do={
     :put $tmpArray;
         /ppp secret add name=[:pick $tmpArray 0] password=[:pick $tmpArray 1] \
             local-address=[:pick $tmpArray 2] remote-address=[:pick $tmpArray 3] \
             profile=[:pick $tmpArray 4] service=[:pick $tmpArray 5];
}
} while ($lineEnd < $contentLen)
```

Detect new log entry

This script is checking if new log entry is added to particular buffer.

In this example we will use pppoe logs:

```
/system logging action
```

```
add name="pppoe"
/system logging
add action=pppoe topics=pppoe,info,!ppp,!debug
```

Log buffer will look similar to this one:

```
[admin@mainGW] > /log print where buffer=pppoe
13:11:08 pppoe,info PPPoE connection established from 00:0C:42:04:4C:EE
```

Now we can write a script to detect if new entry is added.

Code:

```
:global lastTime;

:global currentBuf [ :toarray [ /log find buffer=pppoe  ] ] ;
:global currentLineCount [ :len $currentBuf ] ;
:global currentTime [ :totime [/log get [ :pick $currentBuf ($currentLineCount -1) ]
time   ] ];

:global message "";

:if ( $lastTime = "" ) do={
     :set lastTime $currentTime ;
     :set message [/log get [ :pick $currentBuf ($currentLineCount-1) ] message];

} else={
     :if ( $lastTime < $currentTime ) do={
             :set lastTime $currentTime ;
             :set message [/log get [ :pick $currentBuf ($currentLineCount-1) ]
message];
     }
}
```

After new entry is detected, it is saved in "message" variable, which you can use later to parse log message, for example, to get pppoe clients mac address.

Allow use of ntp.org pool service for NTP

This script resolves the hostnames of two NTP servers, compares the result with the current NTP settings and changes the addresses if they're different. This script is required as RouterOS does not allow hostnames to be used in the NTP configuration. Two scripts are used. The first defines some system variables which are used in other scripts and the second does the grunt work:

```
# System configuration script - "GlobalVars"

:put "Setting system globals";

# System name
:global SYSname [/system identity get name];
```

```
# E-mail address to send notifications to
:global SYSsendemail "mail@my.address";

# E-mail address to send notifications from
:global SYSmyemail "routeros@my.address";

# Mail server to use
:global SYSemailserver "1.2.3.4";

# NTP pools to use (check www.pool.ntp.org)
:global SYSntpa "0.uk.pool.ntp.org";
:global SYSntpb "1.uk.pool.ntp.org";
# Check and set NTP servers - "setntppool"

# We need to use the following globals which must be defined here even
# though they are also defined in the script we call to set them.
:global SYSname;
:global SYSsendemail;
:global SYSmyemail;
:global SYSmyname;
:global SYSemailserver;
:global SYSntpa;
:global SYSntpb;

# Load the global variables with the system defaults
/system script run GlobalVars

# Resolve the two ntp pool hostnames
:local ntpipa [:resolve $SYSntpa];
:local ntpipb [:resolve $SYSntpb];

# Get the current settings
:local ntpcura [/system ntp client get primary-ntp];
:local ntpcurb [/system ntp client get secondary-ntp];

# Define a variable so we know if anything's changed.
:local changea 0;
:local changeb 0;

# Debug output
:put ("Old: " . $ntpcura . " New: " . $ntpipa);
:put ("Old: " . $ntpcurb . " New: " . $ntpipb);

# Change primary if required
:if ($ntpipa != $ntpcura) do={
    :put "Changing primary NTP";
    /system ntp client set primary-ntp="$ntpipa";
    :set changea 1;
```

```
    }

# Change secondary if required
:if ($ntpipb != $ntpcurb) do={
    :put "Changing secondary NTP";
    /system ntp client set secondary-ntp="$ntpipb";
    :set changeb 1;
    }

# If we've made a change, send an e-mail to say so.
:if (($changea = 1) || ($changeb = 1)) do={
    :put "Sending e-mail.";
    /tool e-mail send \
        to=$SYSsendemail \
        subject=($SYSname . " NTP change") \
        from=$SYSmyemail \
        server=$SYSemailserver \
        body=("Your NTP servers have just been changed:\n\nPrimary:\nOld: " . $ntpcura
. "\nNew: " \
          . $ntpipa . "\n\nSecondary\nOld: " . $ntpcurb . "\nNew: " . $ntpipb);
    }
```

Scheduler entry:

```
/system scheduler add \
  comment="Check and set NTP servers" \
  disabled=no \
  interval=12h \
  name=CheckNTPServers \
  on-event=setntppool \
  policy=read,write,test \
  start-date=jan/01/1970 \
  start-time=16:00:00
```

Auto upgrade script

- Auto_upgrade_script_V3.x

Other scripts known to work with latest v3.x

- Dynamic_DNS_Update_Script_for_EveryDNS
- Dynamic_DNS_Update_Script_for_ChangeIP.com
- UPS Script

LUA Scripting examples

NOTE!

After RouterOS v4.0beta4, Lua support is removed until further notice

In v4.0beta3 Lua scripting language is integrated in console. This integration allows users to create their own functions and bypass several command line scripting limitations.

All examples below require at least basic knowledge of Lua scripting language. Good tutorials can be found here as a starting point.

Print function

As stated in Lua documentation, 'print' command is not available in RouterOS compared to standard Lua release. This example will show you how to get back 'print' command

```
-- ------------------------------------------------------------------------
-- Print function
-- ------------------------------------------------------------------------
function print (...)
   local strPrintResult = ""
   if ... then
      local targs = {...}
      for i,v in ipairs(targs) do
        strPrintResult = strPrintResult .. tostring(v) .. "  "
      end
      strPrintResult = strPrintResult .. "\r\n"
      io.write(strPrintResult)
   end
end
```

Now you can include this custom function to other scripts and use this cool custom print function :)
You can also modify this function to write messages in RouterOS log.

Read and write large files

Many users requested ability to work with files. Now you can do it without limitations.
Create and write to file:

```
:global newContent "new file content\r\nanother line\r\n";
[/lua "local f=assert(io.open('/test.txt', 'w+')); f:write(newContent); f:close()" ];
```

Read file content to variable:

```
:global cnt ""
[/lua "local f=assert(io.open('/test.txt', 'r')); cnt=f:read('*all'); f:close()" ];
:put $cnt
```

Include custom function in another script

This example will show where to store and how to include your cool custom created functions into another scripts

- In router's file root directory create subdirectory named 'lua'
- On your PC create new file named customprint.lua and write this function in it.
- Upload newly created file in router's 'lua' directory that we made in first step
- Now you can test your custom lua function

```
[:lua "require 'customprint'\n print('hello from custom print function')"]
```

Scripts

Setup

- A script to set up WAN/LAN/WLAN to get you started
- How to Make an Automated Configuration and Uninstall

General

- A Bit of Sounds
- Automated Billing Script
- Automated Usage Script without usermanager
- Backup graphing data
- Calculate with decimal numbers
- Calea perl trafr
- Converting network and gateway from routing table to hexadecimal string
- Dial PPPoE until a Certain IP Range is Obtained
- Dynamic DNS on private DNS server (Router OS, Bind, Apache, and Shell script)
- Dynamic DNS Update Script for ChangeIP.com
- Dynamic DNS Update Script for ChangeIP behind NAT
- Dynamic DNS Update Script for DNSoMatic.com
- Dynamic DNS Update Script for DNSoMatic.com behind NAT
- Dynamic DNS Update Script for dynDNS
- Dynamic DNS Update Script for dynDNS behind NAT
- Dynamic DNS Update Script for EveryDNS
- Dynamic DNS Update Script for Hurricane Electric DNS
- Dynamic DNS Update Script for Namecheap
- Dynamic DNS Update Script for No-IP DNS
- Email setup/troubleshooting
- Get active VPN connections via e-mail (PPTP and L2TP)
- GPS text file converter to Google Earth/Maps
- Hurricane Electric IPv6 Tunnel - IPv4 Endpoint updater
- IP Pool Statistics
- Log Parser - Event Trigger Script
- Queue tree and e-mailing stats
- Remove BUSY status DHCP Leases to solve malfunction of DHCP server
- Routing via a DHCP allocated gateway (when this address could change and is not a default route)
- Scheduled disconnect for WAN-Interface e.g. DSL
- Scheduled check for loaded interfaces (auto adding queue to some IP or interface)
- Script to create directory
- Script to find the day of the week
- Script to monitor unexpected script failure
- Sending text out over a serial port
- Sending your self an e-mail with DSL interface IP address
- Set global and local variables
- Setting static DNS record for each DHCP lease
- Super Mario Theme
- Traffic Prioritization Script
- Update static DNS entries every 10mins. (Specifically in cases where the upstream ISP "loadbalance" between SMTP servers by using a low TTL on their SMTP DNS)
- UPnP Multi-WAN
- Use Functions in CMD Script
- Use host names in firewall rules
- Use Mikrotik as Fail2ban firewall

- Useful Bash Scripts
- Using 'find' command to filter a command output
- Using scripting to overcome the inability to specify number ranges on the command line
- Wake on Lan before connection to Remote Desktop

Hotspot

- Add a data limit to trial hotspot users
- Enable/Disable new guest user account daily
- Expire users a after number of days
- PayPal with hotspot and walled garden bypass
- Reset Hotspot user count

Modifying Router Settings 'on the fly'

- Blocking Rapidshare.com web page
- Enable and Disable P2P connections
- Enable Disable Firewall Rules
- Generate bogons firewall chain based on routing-marks
- Limiting a user to a given amount of traffic (using firewall)
- Limiting a user to a given amount of traffic II (using queues)
- Limiting a user to a given amount of traffic with user levels (using queues)
- Limit Different Bandwidth In Day and Night
- Random MAC/Ethernet address generate and apply
- Using Fetch and Scripting to add IP Address Lists

Resilience/Monitoring

- ECMP Failover Script
- Improved Netwatch
- Improved Netwatch II
- Failover con Netwatch III
- Failover via Netwatch III (English)
- Easy Failover using only a script
- Force Disconnect Wireless Stations with Low CCQ
- Monitor logs, send email alert / run script
- Monitoring Script
- PPP Keepalive ping
- Secure L2TP server for IPSec clients only
- Send email about reboot

System Maintenance

- Add Static DHCP Leases to ARP List
- Alignment Script that "reads back" RSSI with beeps
- Antenna Alignment with RB532 LED
- Audible signal test
- Auto upgrade script V3.x
- Automated Upgrade/Downgrade script V3.9+
- Automatic Backup with Centralized Storage
- BackupROS (Centralized Backups) - by Nahuel Ramos (new!)
- Batch deployment of DSA key (SSH) and schedule backup with export
- Centralized Automated Backups via Email with Procmail and Perl
- Delete ARP trafic for arp table

- Flash Friendly Backup Script
- Generate routes for stress testing BGP functionality
- Improved auto upgrade script v3.X
- Improved Semi-automatic system-update script
- Logging Average CCQ and Wireless Clients Stats
- Logging SNR and thruput values
- Managing Power Distribution Unit via SMS
- Monitor input voltage on RB333/433AH
- Netwatch on web
- Reboot Boards due to low Memory with notification
- Remotely change password for managers
- Scheduled sending of an email with system backup attached
- Semi-automatic system-update by script
- Semi-Automating CPE ROS/Firmware/script updates and setting changes
- sending mails when on battery or battery low
- Sync Address List with DNS Cache
- Sync Address List from DNS Lookup Results - CNAME and A Records
- SXT 5HnD Alignment Script
- Use SSH to execute commands (DSA key login)
- Yet Another Alignment Script With LEDs And Sound
