Service Provider Solutions

www.virtualaccess.com

Service Managed Gateway Script Language User Guide

Issue 1.3
Date 26 February 2014

1 Introduction

The Service Managed Gateway (SMG) scripting language lets you create script files that can perform a variety of operations, such as:

- dynamically enabling or disabling filters,
- monitoring for changes,
- delivering statistics, and
- running repetitive tests.

Script files can execute immediately, or they can wait for specific events and then perform some action in response to the event.

You can create, run, and otherwise manipulate SMG script files using the script editor or the scheduler in the SMG web, Telnet, or the router console if your model has one.

You write script statements using script commands, variables, comparative expressions, and embedded commands. You manipulate scripts from the console using non-script console commands.

Advanced scripting features expand the script creation options that are available to you.

The audience of this document are customers' engineers and Virtual Access' engineers, who might need to implement a script in the SMG.

2 Using the script editor

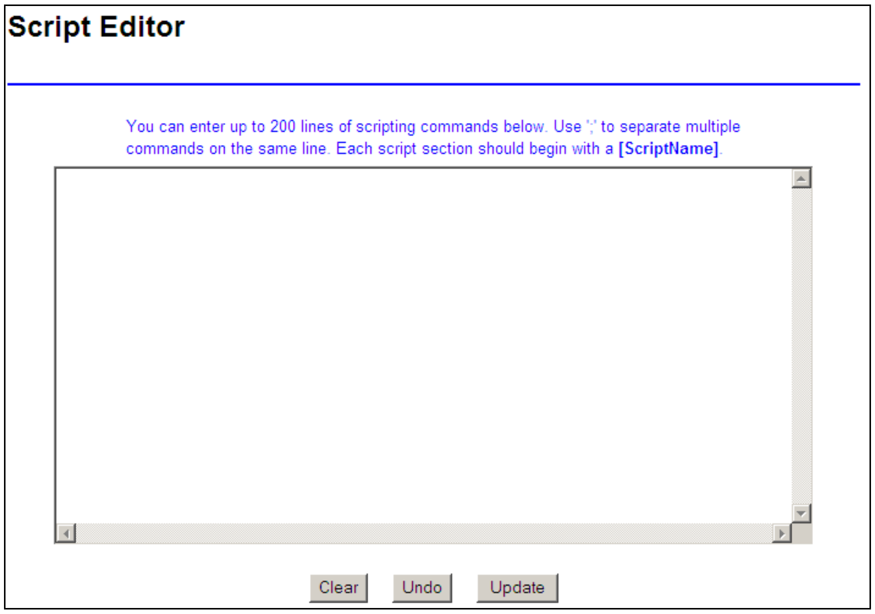
The Script Editor is the primary tool for creating script files. When you create a script in the Script Editor, a permanent copy of the script is saved in the router's configuration file.

To open the Script Editor, on the Start page, click **Advanced**.

In the Advanced menu, click **Expert View**.

If you are using 9.09.xx firmware, in the Expert view menu, select **system > scripts > script editor**. The script editor appears.

If you are using 10.00.xx firmware, in the expert menu, select **system > management > scripts > scripts editor.** The script editor appears.



You can enter up to 200 lines of scripting commands below. Use ';' to separate multiple commands on the same line. Each script section should begin with a **[ScriptName]**.

Figure 1: The script editor

You write script statements using script commands, variables, comparative expressions, and embedded commands. A line of script may contain more than one script command statement, and you must separate each command statement with a semicolon (;). Lines can contain up to 140 characters. You can store up to 200 lines of script in the script editor.

The script editor can contain one or more actual scripts. Each script begins with the notation:

```
[scriptname]
```

When you are executing the script later on, you refer to it by this name.

For example, two scripts that watch the PPP link and VPN tunnels might look like this:

```
[watchppp]

!while 1

!waitevent ppp:ncp_up

test email Your PPP link just came online.

!endwhile
```

```
[watchvpn]

!while 1

!waitevent ike:Connected

test email Your VPN link just came online.

!endwhile
```

The use of the [scriptname] syntax to introduce scripts in the script editor makes it easy to allow several scripts to co-exist in the same configuration file; these are conceptually treated as individual scripts.

2.1 Buttons in the script editor

There are three buttons in the script editor.

Clear: deletes the entire contents of the edit buffer. Use it when you are about to start entering a new script or set of scripts.

Undo: removes any changes you have made since the page originally loaded. It also restores the content if you accidentally click Clear. If you accidentally click Undo, click it again; it will restore what was undone.

Update: updates your router configuration with the changes that you just made.

3 Writing scripts

Scripts are composed of command statements, which you build using script commands and parameters. You can also include comments at the start of a line in a script, comments begins with //.

Command statements within scripts function like ordinary statements entered into the router from the console. The results of the commands are echoed back to the console.

For example, the script:

```
!var port=1

!while $port<9

 show ip if addr ppp-$(port)

 !add $port, 1

!endwhile
```

will execute the following router commands:

```
show ip if address ppp-1

...

show ip if address ppp-8
```

3.1 Script commands

Script commands allow a script to loop, produce output, delay, make decisions, and so on. They can only be executed from within a script file.

When a script command modifies the value of a variable, do not begin the variable name with $. The commands in this category are !inc, !dec, !var, !add, and !sub.

The following script commands are available.

3.1.1 !ADD

```
!add <variable>, <value>
```

This command adds <value> to the current value of <variable>. <value> and must be a numeric value.

When you use the !add command, remember to leave out the $ at the start of the variable name.

3.1.2 !ARG

```
!arg <param1>, <param2>,
```

Use this command to give more friendly names to the parameters passed to a script. By default, parameters are named $1, $2, $3, etc. When an !arg command is encountered, $1 is assigned to <param1>, $2 is assigned to <param2>, and so on.

If too few parameters were passed to the script, a helpful error message appears listing the expected argument names and the script terminates.

3.1.3 !CALL

```
!call <scriptname> [ parameters ]
```

This command executes the commands in the script `<scriptname>`. Unlike the `run` command, which runs scripts in the background, `!call` will wait until all the commands in `<scriptname>` have completed before continuing.

3.1.4 !DEC

```
!dec <variable>
```

This command reduces the current value of `<variable>` by one.

When you use the `!dec` command, remember to leave out the $ at the start of the variable name.

3.1.5 !DUMPVAR

```
!dumpvar [ <matchstring> ]
```

Use `!dumpvar` to display a list of all the variables defined in the current script, along with their values. This is primarily of use for debugging. If the optional matchstring is given, then only variable names matching that string are printed.

3.1.6 !ECHO

```
!echo on | off
```

This command controls whether or not the script displays router commands before they are executed. !Echo is off by default, but it can be useful to turn it on for debugging purposes to confirm that the parameters passed to a command are as expected. Any parameters other than On or Off are written to the current output device.

3.1.7 !EVENT

```
!event <class>.<subclass>.<ifindex> [ <message> ]
```

Use `!event` to generate a system event of the given `<class>` and `<subclass>` with interface number `<ifindex>`. `<class>` may be specified as either a numeric event code or an event name, while `<subclass>` is always numeric. If `<subclass>` is omitted, then it defaults to 0, as does `<ifindex>`. `<ifindex>` may also be specified as an interface name, but you cannot use abbreviations.

Most events have a string associated with them already. If you specify `<message>`, then this will be appended to the existing event string, if there is any, when the event is generated.

Note: a list of event names can be displayed by typing show `event classes`, and a list of all the recognized event codes can be displayed by typing show `event table`.

3.1.8 !EXIT

```
!exit
```

When this command is executed, the script halts execution immediately. If the script was called from another script, then the other script resumes execution.

3.1.9 !GOTO

```
!goto <label>
 ….
!label <label>
 ...
```

This pair of commands provides a simple mechanism for transferring control from one part of the script to another. When the `!goto` command is issued execution continues at the line containing the statement `!label <label>` .

3.1.10 !IF

```
!if <expr> <comparisontype> <patternexpr>
Block A
!else
Block B
!endif
Block C
or
!if <expr> <comparisontype> <patternexpr>
Block A
!endif
Block C
```

If the comparison evaluates to true, Block A (which is any sequence of commands) is executed and then Block C is executed. If the comparison evaluates to false, in the first case Block B is executed and then Block C is executed. In the second example, as there is no Block B (`!else`), only Block C is executed.

If both expressions are numeric, the comparison is done numerically; otherwise, it is a case-insensitive string comparison. If the right-hand side of the comparison is a string beginning with '*', then it is treated as a wildcard pattern.

When checking the value of a variable with !if, it is usually wise to omit the leading $ from the variable name. Otherwise, the variable name is replaced with its current value before executing the !if statement, and this can lead to problems if the variable value is empty.

Note: current implementation requires a space on either side of `<comparisontype>`.

For further information about pattern expressions, read section 3.2.7 'Pattern matching'.

3.1.11 !INC

```
!inc <variable>
```

This command adds one to the current value of `<variable>`.

When you use the `!inc` command, remember to leave out the $ at the start of the variable name.

3.1.12 !LOG

```
!log <message>
```

This command writes <message> to the system event log as a script event. This is a convenient way to record what the script is doing.

3.1.13 !PAUSE

```
!pause <value>
```

This command pauses script execution for <value> seconds.

3.1.14 !PRINT

```
!print <string>
```

This command prints the characters in string and adds a newline character. All variables are replaced by their values before printing.

3.1.15 !SPEED

```
!speed <iterationcount>
```

By default, the script interpreter will execute at most 10 script commands per timer tick before pausing to give the router time to perform other operations, such as routing. Running an external command or pausing to wait for an event or timeout can reduce this. Timer ticks happen every 0.2 seconds, so at most the script will execute 50 commands per second.

If a script is carrying out a large number of actions, for example, scanning all the SPD entries in a VPN configuration, faster operation may be required. The `!speed` command allows a larger number of commands per timer tick to be iterated. If the value specified is too large, the router may introduce a delay in routing normal packets, so this should be treated with caution. The default value is `!speed 10`.

As mentioned, running an external command will cause a script to pause immediately after the external command has completed executing. The script will resume when the next timer tick occurs. This is because external commands can often require significant processing. However, this can be inconvenient with simple external commands, since it can make scripts run more slowly. To avoid such delays, external commands can be run as command expressions, like this:

```
$dummy = ‘show ip interface address e0‘
```

Because the command is executed as an expression, the scripting engine will not pause afterwards, but will treat the command in the same way as a normal script command.

3.1.16 !SUB

```
!sub <variable>, <expr>
```

This command subtracts <expr> from the current value of <variable>. <expr> must be a numeric value.

When you use the !sub command, remember to leave out the $ at the start of the variable name.

3.1.17 !UNIQUE

```
!unique
```

Some scripts are designed to run indefinitely in the background, performing an operation periodically. A ping test is a good example of this. If such a script needs to be restarted, you can end up with two or more copies running simultaneously. This is undesirable, since they may end up doing the same actions multiple times.

To avoid this, include the !unique command near the start of the script. When the script is executed, any other copies of the same script that are currently executing will be terminated, and only the current script will remain running. Other, unrelated, scripts that may be running are not affected.

3.1.18 !VAR

```
!var <variable> = <value>

<$variable> = <value>
```

Assigns <value> to <variable>.

You can omit the script command !var and write <variable> = <value>. However, if you do this, you must begin the variable name with $.

3.1.19 !WAITEVENT

```
!waitevent <class>.<subclass>:<patternexpr> [ <delay> ]

Block A

!endevent

Block B
```

This command waits until the event specified by <class> and <subclass> has occurred. These are the standard events normally written to the router's event log or displayed on the console. You can specify <class> as a number or as an event name. <subclass> is optional—if it is not specified, then any event of type <class> will satisfy the wait condition. Specifying a subclass of 0 has the same effect.

If the optional text <patternexpr> is specified, then <patternexpr> must occur in the description of the event in order for a positive match to occur. This works even if you did not specify a specific <subclass>. This is useful where you need to wait for a specific event string that has not been assigned a unique <subclass> code within its event group. <patternexpr> can be a simple text string or a more complex pattern, as described earlier. In a simple text string, use underscores (_) to represent spaces.

For more information about pattern expressions, read 3.2.7 'Pattern matching'

If the event occurs within `<delay>` seconds, Block A executes, followed by Block B. If the event has not occurred within the defined time, then Block A is skipped and execution continues immediately with Block B.

If no delay is specified, then the script will wait indefinitely for the given event, until it is killed using the `KILL` command. As mentioned earlier, the command `show event table` will list all recognized events.

For convenience, `!we` is recognized as a valid abbreviation for `!waitevent`. This is useful when executing scripts directly from the command line.

The following special variables are defined whenever a `!waitevent` command completes:

`$EvClass`	The class number of the event
`$EvSubclass`	The subclass number of the event
`$EvIndex`	The interface or logical index associated with the event
`$EvText`	The text string associated with the event

You can use the `$EvIndex` variable to identify the router interface associated with an event. Each interface is assigned an SNMP index value, which can be seen by executing the router command `show interfaces`. For example, eth-0 is interface 1, while ppp-8 is interface 16.

Not all events relate to interfaces; in such cases, `$EvIndex` will hold a list index value corresponding to the event. For example, VPN events typically set `$EvIndex` to an IKE or SPD policy number. For advance settings on VPN check the VPN configuration guide.

Use the router command `show interfaces` to get a full list of interfaces.

3.1.20 !WHILE

```
!while <expr> <comparisontype> <patternexpr>

Block A

!endwhile

Block B
```

`!while` is the main loop type allowed in the language. As long as the condition after the while evaluates to true, Block A will be executed. Once the condition is false, execution continues with Block B. Remember that `!ew` is a valid abbreviation for `!endwhile`.

When you use the `!while` command, remember to leave out the `$` at the start of the variable name.

Note: Infinite loops must be halted with the `KILL` command.

3.1.21 !OPENCONNECTION & !CLOSECONNECTION

```
!openconnection <ipaddress> <portnumber> connectionID (returned),
result(returned)


!closeconnection <connectionID>
```

`!openconnection` opens a TCP connection to an IP address on a particular port. This can be used to communicate with applications such as terminal server on the SMG. If Telnet is specified the SMG will negotiate Telnet parameters. If the connection attempt is successful then it will return a connectionID. A connection attempt will always return a `result`, 0 for a failed connection attempt and 1 for a successful connection attempt. `connectionID` and `result` must be declared as variables before using the `!openconnection` command.
When using the `!openconnection` command, remember to leave out the $ at the start of the variable name.
Note: if the `!closeconnection` command is not used then the telnet connection will stay open until it is terminated by the server.

3.1.22 !READSTRING

```
!readstring <connectionID> <stringvariable> <bytesread> result (returned)
```

This command allows a script to read a string from a TCP connection. `stringvariable` and `bytesread` should be declared as variables.
This command also returns a result based on the success of the command's read attempt. If a 0 is returned this is a fail, if a 1 is returned this means a success.
If using the `!readstring` command it should be used regularly. Otherwise the data that is read will be discarded once the buffer reaches it's limit.

3.1.23 !WRITESTRING

```
!writestring <connectionID> <stringvariable>
```

This command allows a script to write a string to a TCP connection. The `connectionID` is returned when the TCP session is established. `stringvariable` should be the string that needs to be written to the TCP connection.

3.1.24 !SNMPTRAP

```
!snmptrap <destinationip> <trapcommunity> <generictrap> <specifictrap>
<variablecount> <oids> <types> <descriptions>
```

This command allows a script to generate an SNMP trap. The `oids, types` and `descriptions` are all arrays. They should all be declared as variables.
e.g.

```
$oid(1) = 1.3.6.1.4.1.2078.10.1
$oid(2) = 1.3.6.1.4.1.2078.10.2

$type(1) = 2 // int
$type(2) = 4 //string

$value(1) = 22
$value(2) = aabbcc

!snmptrap 10.1.1.2, public, 6, 0, 2, oid, type, value
```

3.1.25 Special Variables

3.1.25.1 $timer

Any script variable called $timer or $timer## (with a number after it, e.g. $timer1, $timer2, $timer3, etc.) is an automatic timer variable. By default, they count time since the router was last rebooted.

You can set a timer to zero by with the following command: $timer = 0; it then increments automatically at the system clock rate.

So, for example:

```
   $timer = 0
   !pause 2
   !echo $timer
```

This should print "2.0123" seconds or something close to that on the screen. This is useful when you need to know the time between different events.

3.2 Script expressions

The parameters used in script commands are defined as follows:

- <variable> is a variable
- <value> is a string of characters or a 32-bit decimal number
- <patternexpr> is a <value> or a defined variable that is matched against an arbitrary text string.
- <expr> is either a <value> or a defined <variable>
- <delay> is a period in seconds

<comparisontype> is one of the following operators:

• <>	• not equals

• ==	• equal to
• >	• greater than
• <	• less than
• >=	• greater than or equal too
• <=	• less than or equal to
• ~	• regular expression

3.2.1 Variables that are used with script commands

A variable is used to hold a value that may change during the execution of the script. Before variables are referenced, they must be declared using the `!var` command or by assigning them an explicit value. Variables can hold either numeric or string values. Strings are restricted to 160 characters in length; however, there is no limit to the number of variables a script may define.

Anything following a `$` character up to the first non-alphanumeric character is taken as a variable name. This is usually how you want things to work since it is rare that you embed variable names inside other strings. However, you can surround the variable name in parenthesis to make it explicit where the name ends. So, these are all valid variables:

```
$1    $test     $(ppp_port)    $(x)
```

Some variables are predefined for you when a script begins executing. `$0` is the number of command line parameters passed into the script, while `$1` … `$9` give the values of the first nine parameters if they are defined. If they are not defined it gives empty. You can use the `!arg` command to rename these parameters to more user-friendly names.

You can also use the `$` notation to reference external router symbols, which have been defined with the router's `set symbol` commands.

Symbol values defined with the router `set symbol` command are saved as part of the router's configuration file. Their values persist across multiple script invocations and even system restarts. They provide a convenient mechanism to allow a single script to behave in different ways, simply by changing the value of the symbols it refers to.

Where both a symbol and variable share the same name, the variable takes precedence. Also, router symbols can only be read, not modified. However, you may modify them indirectly by issuing `set symbol` router commands from within the script.

3.2.2 Naming variables

The commands `!add`, `!sub`, `!inc`, `!dec`, `!if`, and `!while` expect a variable name as one of their parameters. When used with `!add`, `!sub`, `!inc`, and `!dec`, this variable name should not generally begin with the variable reference `$`.

You should not use `$` to begin a variable reference because the scripting engine replaces all `$` variable references with their equivalent value before it actually executes the command. So, if `$y` is 4, the expression "`!add x, $y`" is translated to "`!add x, 4`" before the `!add` command runs. This means that "`!add $x, $y`" could be translated to "`!add 0,4`" which is meaningless because there is nowhere to store the result.

The commands `!add` and `!sub` do allow a variable reference when there is a second value in the string, so you can begin the second variable name with $. For example, `!add x,$y` adds the current value of y to the variable x.

You can also use a variable reference with `!if` and `!while` when you want to combine variables. For example:

```
!if $x.$y = “ppp-1.input”

      …

!endif
```

checks the value of both x and y at the same time.

3.2.3 Variable subscripts

It is often useful to assign the output of a command to a variable and then reference portions of that output. This can be done using subscripts. There are two types of subscript: indexing and ranging. Indexing lets you treat the variable as a list of discrete elements and extract one individual element. Ranging lets you take a subset of the variable’s characters.

To index a string, use the syntax `$var[index]`. By default, elements are assumed to be separated by either a comma or a tab. So, if `$var` is “first,second,third”, `$var[1]` will be “first”, `$var[2]` is “second” and `$var[3]` is “third”.

If you want to use a different separator character, such as =, you can specify this in the reference using a colon. For example, if `$var` is “time = 10:34”, then `$var[1:=]` will be “time” and `$var[2:=]` is “10:34”. You can apply multiple subscripts in sequence if you wish, so `$var[2:=][1::]` will first calculate the substring “10:34”, then split this into a second list of elements separated by ‘:’ and finally return “10” on its own. Note that you can specify multiple separation characters in the list, though usually you will not need to.

A typical example of an index is to extract the elements of an IPAT table entry:

```
var $ipat = ‘find 1.2.3.4 ipat‘

echo Port = $ipat[0], IP address = $ipat[1], etc….
```

This can also be useful if you need to repeat a loop on a range of values:

```
$iflist = ppp-1,ppp-2,ppp-3,ppp-4,ppp-5,ppp-6,ppp-7,ppp-8

$i = 1

!while i <= 8

$interface = $iflist[$i]

// Do things with $interface

!inc i

!endwhile
```

3.2.4 Variable ranges

To specify a range, use the syntax `$var{x-y}` where x and y are character positions. The simplest way to demonstrate this is with some examples. In all these cases, `$var` has the value “hello world”.

`$var{1}`	“h”	(extract a single character)
`$var{7-11}`	“world”	(straight substring)
`$var{-3}`	“rld”	(last three characters of the string)
`$var{2-}`	“ello world”	(everything from the second char on)

Ranges and indexes can be freely mixed on a line, so `$var[2]{3}` returns the third character of the second element. In all cases, any leading or trailing spaces present are removed after the final string has been prepared.

Range parameters must be numeric; it is not currently possible to use variables to define the range.

3.2.5 Variable arrays

Arrays are used to allow one variable to store a range of values, each associated with a particular index. An array index is represented by simply putting it in parenthesis following the variable name. For example:

```
$count(0)= 10

$count(1)= 20

$i              = 1


!print $count($i)       -- prints 20
```

Arrays are associative, in that they can use both numbers and strings as indices. For example, `$total(ppp-1)` is valid.

3.2.6 Comparative expressions

When a statement compares two expressions, numeric values are compared numerically, while strings are compared as case-insensitive text. In addition, if the right-hand side of a string comparison begins with a `*`, then the remainder of the string is treated as a wildcard pattern. For more information about this, read section 3.2.7 ‘Pattern matching’

Any command that expects a comparative expression can also accept a single <exp> on its own. This is treated as an implicit comparison against zero. For example, you can say !while1 to create an infinite loop. Simarly, `!if $variable` is sufficient to check whether $variable has been set to a real value, or is empty.

3.2.7 Pattern matching

The commands `!waitevent, !if,` and `!while` can accept a pattern string that is matched against an arbitrary text string. If the comparison is successful, some action takes place. The pattern string can be any of the following:

`abc`	True if the text string contains ‘abc’
`abc|def`	True if the text string contains ‘abc’ or ‘def’
`abc&def`	True if the text string contains both ‘abc’ and ‘def’
`abc&!def`	True if the text string contains ‘abc’ but *not* ‘def’

Any number of each operator can be used. Note that ‘`!`’ has the highest precedence, followed by ‘`&`’ and then ‘`|`’. In simple terms, this means that negation takes priority over ‘and’, which takes priority over ‘or’.

For example, `!abc&def|ghi` will match strings that contain either:

```
"def" and not "abc"
ghi
```

If you want to match strings that do not contain abc and either `def` or `ghi`, you need to use `!abc|!def&!ghi`.

If you need to represent a space in the match string, use an underscore (_) . This will automatically match any spaces in the text string.

When you use pattern matching as part of an `!if` or `!while` command, the pattern must always be the second string in the expression.

When you specify strings, you should use quotes only in the context of an `!if` or `!while` statement, where they can be present on both sides of the comparison. This is essential if one or both values being compared may include spaces. For example, these are equivalent comparisons:

```
!if color = "red"
!if "$color" = "red"
```

This presumes that a variable called '`color`' has been defined.

3.2.8 Regular Expressions

The scripting language supports a basic implementation of regular expressions. This allows matching strings of text, such as particular characters or words. For example:

```
!if $st ~ ^.........G
```

The ~ represents the use of a regular expression.
The ^ represents the start of the string.
Each dot `..........` represents a character (any character) in the string.
`G` represents the character being looked for in the regular expression. The character(s) can be located at any point in the string.

3.2.9 Embedded commands

If a command line contains a portion enclosed in back quotes ('like these') then everything between the quotes is treated as an embedded command and executed first. The output from this embedded command is then inserted back into the original command line, which is then executed as normal. For example, the command:

```
echo Your IP address is 'runshow ip interface address ppp-1'
```

may display something similar to "`Your IP address is 159.134.237.6`" when it is executed.

The back quote character is often found in the top left corner of the keyboard.

3.2.10 Expression Evaluator

The SMG has its own expression evaluator. It performs a mathematical calculation on one or more attributes. The command is simply called, **eval** and you can follow it with any numeric or string expression. For example:

```
eval 1+2+3+4
 eval 2*(3+4)
 eval 3/(4+5)
 eval 37.2 * 49.6
```

eval "There are" + 45 + " items in the bag."
It supports three types of values: string literals, integers, and floating point. Values are converted to/from each type as needed, so for example, adding a number to a string concatenates the string representation of that number to the existing string.

There are a small set of built-in functions for doing explicit conversions:

For example:

@ABS	Converts a value to an absolute value
@INT	Converts a value to an integer
@FLOAT	Converts a value to a Float
@STRING	Converts a value to a string

eval "The absolute value is " + @ABS(-43)

Supported operators are:

- **Arithmetic:** +, -, *, /, %
- **Boolean logic:** !, &&, ||
- **Bitwise logic:** &, |
- **Comparisons:** <, <=, ==, !=, >=, >
- **Grouping:** (..)

String literals must be quoted if they include spaces or other non-alphanumeric characters.

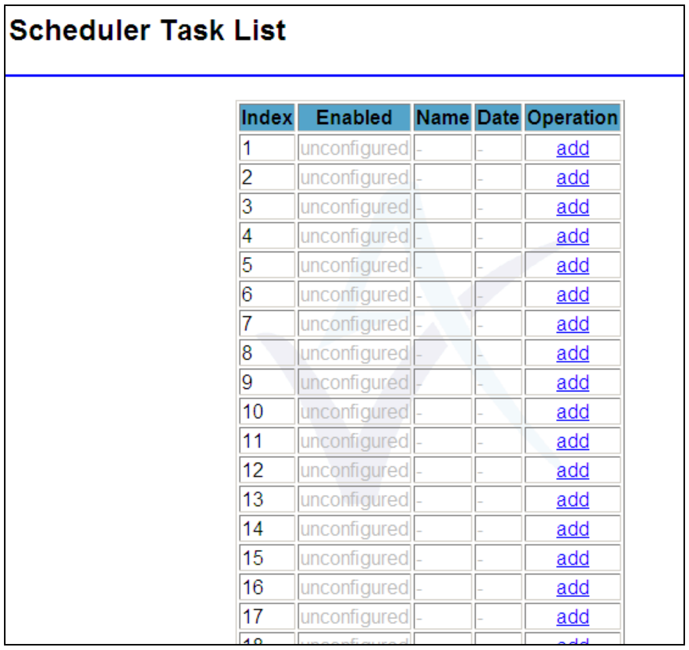
No variables are supported directly, but when calling from the scripting language, you just pass them in as $values directly.

3.3 Scheduling script execution

The SMG has a feature that allows the user to schedule the execution of a script. The scheduler can be used to run a small script defined directly on the scheduler form, or, the most common use of the scheduler, is to trigger the execution of a more sophisticated script that has been created with the script editor

If you are using 9.09.xx firmware, to use the scheduler, in the Expert View menu, select **system > scheduler > scheduler tasks**. The Scheduler Task List appears.

If you are using 10.00.xx firmware, to use the scheduler, in the Expert View menu, select **system > management > scheduler > scheduler tasks**. The Scheduler Task List appears.

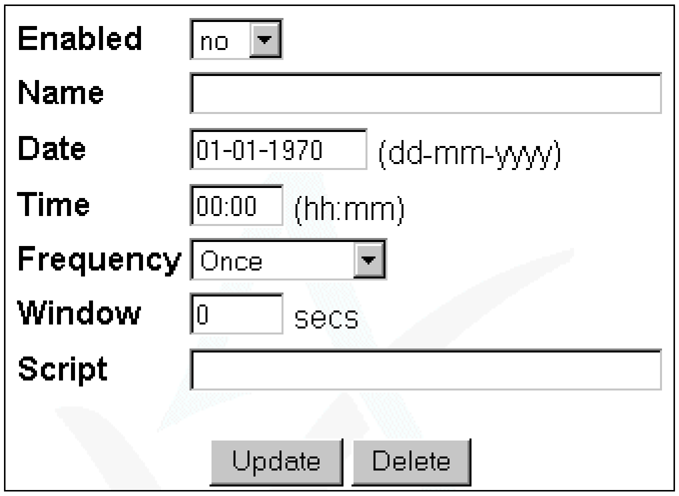


Scheduler Task List

Index	Enabled	Name	Date	Operation
1	unconfigured	-	-	add
2	unconfigured	-	-	add
3	unconfigured	-	-	add
4	unconfigured	-	-	add
5	unconfigured	-	-	add
6	unconfigured	-	-	add
7	unconfigured	-	-	add
8	unconfigured	-	-	add
9	unconfigured	-	-	add
10	unconfigured	-	-	add
11	unconfigured	-	-	add
12	unconfigured	-	-	add
13	unconfigured	-	-	add
14	unconfigured	-	-	add
15	unconfigured	-	-	add
16	unconfigured	-	-	add
17	unconfigured	-	-	add

Figure 2: The scheduler task list

Click **add** in the Operation column of the list. The Scheduler Task form appears.



Enabled	no
Name	
Date	01-01-1970 (dd-mm-yyyy)
Time	00:00 (hh:mm)
Frequency	Once
Window	0 secs
Script	

Figure 3: The scheduler task form

Enable this scheduler task entry setting up Enabled field to **yes**.

In the name field, give it a name. The name doesn't have to follow any particular convention, but we recommended you use a descriptive name for the task.

Then specify the Date and Time you want the script to be run.

The Frequency field sets the frequency at which the script will be executed, it can take the following values:

Once: The script will execute once at the specified date and time.

Startup: The script will execute at every startup regardless of time and date set.

EveryDay: The script will execute everyday at the specified time.

Sunday: The script will execute every Sunday at the specified time.

Monday: The script will execute every Monday at the specified time.

Tuesday: The script will execute every Tuesday at the specified time.

Wednesday: The script will execute every Wednesday at the specified time.

Thursday: The script will execute every Thursday at the specified time.

Friday: The script will execute every Friday at the specified time.

Saturday:The script will execute every Saturday at the specified time.

The Window field specifies how long the system will wait if it is busy before executing the script. For example, if the script is set to execute at 10:00 and the window is set to 30 seconds, the system will try to execute the script within this window only.

Finally, in the Script field you can specify a small script or the name of a more complex script defined using the script editor to be executed.

If you want to define a small script to be scheduled directly on the scheduler task form, you can enter script commands directly in the Script field.

You can enter up to 140 characters into the Script field. You can also enter a mix of script and non-script console commands, separated by semicolons. For more information about non-script console commands, read section 5, 'Using advanced scripting features'.

Scripts you create using the Scheduler Task form persist indefinitely as part of your configuration. They are executed according to the schedule you specify in the form.

If you want to use the scheduler to trigger a more sophisticated script that has been created with the script editor, the most common use of the scheduler, you have to specify the name of the script you want to execute in the Script field.

4 Executing scripts

In general, you manipulate scripts from the console. Type the script name at the command line of the console. The script will execute in the background. Any output appears on the same console.

4.1 Passing parameters to a script you want to execute

You can pass parameters to a script when you execute it from the console. Type the parameters after the script name and separate multiple parameters with commas.

4.2 Manipulating script files from the console

So far, we have seen that scripts can be created using the script editor and executed using the task scheduler. For more advanced purposes, scripts can also be stored as external files in the router's flash file system. Scripts stored in this way can be executed in the same fashion as scripts stored in the router's configuration. In addition, there is no limit to the number of lines an external script file can contain.

The commands you use to create and manipulate external script files are described below, along with a number of general commands used to view currently executing scripts, kill scripts, and assist with script debugging.

The commands also let you manipulate script files from the console. Type the command and the script filename at the command line of the console.

In the descriptions below, <command> refers to any valid command sequence. It may include a mixture of both script and non-script console commands, separated by semi-colons (;).

4.2.1 Show

```
Show <scriptname>
```

Displays the contents of an existing script.

```
Show Config Script All
```

Lists the names of all scripts defined in the current configuration file.

```
Show Config Script <scriptname>
```

Prints the contents of that script, neatly formatted and indented, one entry per line, on the console.

```
Show Tasks
```

Shows all running script tasks on the router, along with their TaskIDs.

```
Show Task [<taskid>| <taskname>]
```

Shows the script currently being executed by the given task. The <taskname> is the same as the scriptname. The line currently being executed is also highlighted.

```
Show Task Vars [<taskid>| <taskname>]
```

Shows all the variables and their values that are currently defined by the executing task.

4.2.2 Kill

```
Kill [all|this| <taskid>| <taskname>]
```

Kills all tasks, tasks associated with this console session, or the task with the listed `<taskid>` or `<taskname>`.

4.2.3 Run

```
Run <scriptname>
```

Runs the script file `<scriptname>` according to the script rules below. Script files execute in the background, so you can run several simultaneously if you wish.

4.2.4 Every

```
Every <n> <command string>
```

Executes `<command string>` every N seconds. `<command string>` can contain any valid scripting commands, just like the scheduler task. The task can be viewed on Show Tasks and stopped with Kill This.

4.2.5 After

```
After <n> <command string>
```

Executes `<command string>` after N seconds. The pending task can be viewed on Show Tasks and stopped with Kill This.

4.2.6 Exec

```
Exec <command string>
```

Executes `<command string>` immediately.

4.2.7 Flashdel

```
Flashdel <filename>
```

Deletes the file from Flash.

4.2.8 Console

```
Console <command>
```

Executes <command> and directs any output to the system console instead of the current telnet or web session. Scripts started in this way do not terminate automatically if the current session closes; they must be killed explicitly or exit of their own accord. Console can also be used inside a quiet script to force some output to be written to the console.

4.2.9 Quiet

```
Quiet <command>
```

Executes <command> and discards any output produced. This is useful for running periodic scripts where you want to avoid cluttering up the console with standard router output messages.

5 Advanced script creation commands

You can create permanent script files over Telnet using the CREATE command. These files are stored in flash memory and will always be available, regardless of the router's current configuration. To create or download scripts, type the commands that are listed below at the command line of the console.

5.1 Create <filename>

`Create <filename>` allows you to create a script file using the console. To create the file:

1. type `Create <filename>` at the command line,
2. paste in the lines of the file,
3. on a line by itself at the end of the file, type a full stop (.), and
4. press Enter.
   - The file is closed and saved to flash. If you type .x at the end of the file, the file is discarded instead of saved. By convention, script files are named with a .bat suffix.

5.2 Dir *.bat

`Dir *.bat` displays a list of all the scripts that are in flash.

5.3 Download <remotefilename> <localfilename>

`Download <remotefilename> <localfilename>` allows a script file to be downloaded from a remote TFTP server and written to flash.

5.4 Flashdel <filename>

`Flashdel <filename>` deletes a script file from flash. Be careful using this command because it can delete system files.

5.5 Frename <oldfilename> <newfilename>

`Frename <oldfilename> <newfilename>` changes the name of a script file.

6 Using advanced scripting features

6.1 The purpose of advanced scripting features

Advanced scripting features expand the script creation options that are available.

6.2 Advanced scripting commands

There are a number of ways you can manipulate scripts more flexibly using Telnet.

6.2.1 Chaining together commands

You can chain together commands. For example:

```
Console every 5 quiet ping 10.1.1.1
```

will start a periodic ping on the console every five seconds and discard the output.

6.2.2 Shortcuts that let you manipulate scripts more quickly

If a command passed to an every or after command is prefixed by @, then the echo of that command name on the screen is suppressed when it is executed. This is similar to setting !echo off, but quicker.

Any standard command line that begins with an ! or $ is automatically treated as if it were prefixed by exec; this can be used to execute short scripts immediately by simply typing them on the command line of the console.

If you create a script file with a `.bat` suffix, it can be invoked automatically by simply typing its name at the command line. For example, a file called TESTALL.BAT can be started by simply typing TESTALL. You do not need to type the Run command or the .bat suffix.

When the router is starting up, it checks for a script file with the name autoexec.bat. if the file exists, then this script is automatically executed. This provides a configuration-independent alternative to the more common approach of using the task scheduler to automatically run a script at startup.

6.2.3 Internal scripts

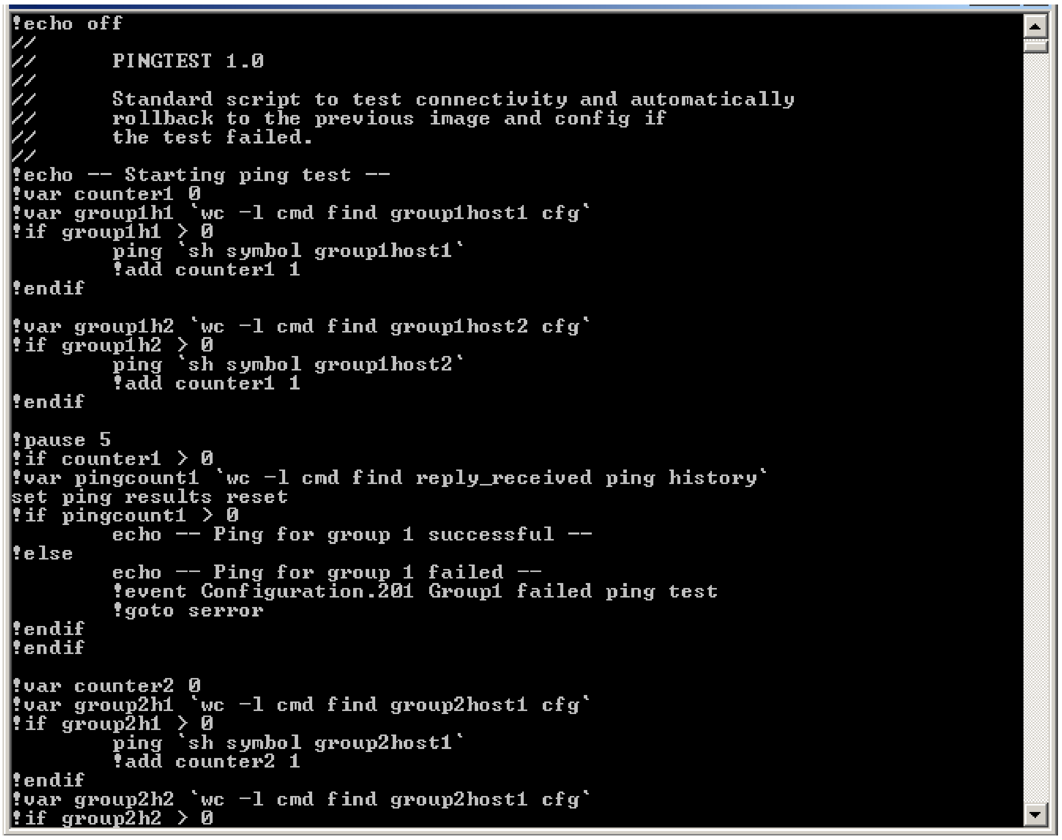
Internal scripts are compiled into the main image. These behave just like scripts stored in flash except that you cannot modify them, only display or execute them. Normally, script files stored in flash take priority over internal scripts. However, it is sometimes useful to force execution of an internal script. You can do this by preceding the script name with the '%' symbol. This will force the script to be executed from the internal script list only, bypassing flash files and library scripts entirely.

To display the directory of internal script files, type '`Dir Scripts`' at the command line.

6.2.4 Using Telnet to create and execute scripts

You can manipulate scripts from Telnet by typing the name of the script at the Telnet prompt.

If you start a script executing from a telnet session, the script will automatically be terminated when the telnet session ends, even if the script has not yet completed. To avoid this, prefix the script invocation with the command console, as described in the section 4.2, Manaipulating script files.



```
!echo off
//
//      PINGTEST 1.0
//
//      Standard script to test connectivity and automatically
//      rollback to the previous image and config if
//      the test failed.
//
!echo -- Starting ping test --
!var counter1 0
!var group1h1 `wc -l cmd find group1host1 cfg`
!if group1h1 > 0
        ping `sh symbol group1host1`
        !add counter1 1
!endif

!var group1h2 `wc -l cmd find group1host2 cfg`
!if group1h2 > 0
        ping `sh symbol group1host2`
        !add counter1 1
!endif

!pause 5
!if counter1 > 0
!var pingcount1 `wc -l cmd find reply_received ping history`
set ping results reset
!if pingcount1 > 0
        echo -- Ping for group 1 successful --
!else
        echo -- Ping for group 1 failed --
        !event Configuration.201 Group1 failed ping test
        !goto serror
!endif
!endif

!var counter2 0
!var group2h1 `wc -l cmd find group2host1 cfg`
!if group2h1 > 0
        ping `sh symbol group2host1`
        !add counter2 1
!endif
!var group2h2 `wc -l cmd find group2host1 cfg`
!if group2h2 > 0
```

Figure 4: Scripts in the command line interface

Appendix A: Scripts included with your Service Managed Gateway

The following internal scripts are built into the Service Managed Gateway and can be executed at any time:

PINGTEST.BAT checks the connectivity to a remote site and reverts to the previous configuration if the connectivity test fails.

VPNTEST.BAT ensures that all configured VPN tunnels can be correctly established and notifies Activator of success or failure for each tunnel.

WATCHPPP.BAT sends an email with the assigned IP address whenever the PPP connection is established.

WATCHSPD.BAT monitors VPN tunnel operation and switches to a backup tunnel whenever connectivity is lost.

BACKUP.BAT is used to switch to a backup interface when a primary interface fails. The backup and primary interfaces must be running ppp.

WATCHDOG.BAT is used dynamically by the BACKUP.bat script in backup mode. It checks the PPP state of the primary interface to ensure that NCP Up on the primary has not been missed. If the interface is found to be UP a dummy 'NCP up'.

PHY_TOGGLE.BAT forces a manual disconnect and reconnect of a physical interface.

As mentioned earlier, you can see those scripts, including a description of what they do, how to use them and some usage examples, running the `Show <scriptname>` command on the command line interface (telnet).

To invoke one of these scripts at startup, configure a scheduler entry using the Scheduler Task form. In the form, select **Startup** in the Frequency drop-down list. In the Script field, type the name of the script you want to run.

You can also execute these scripts by typing the file name at the command line of the console.

To list all available internal scripts, use the command `dir scripts`.