



Using the CSS Scripting Language

The CSS includes a rich Scripting Language that you can use to write scripts to automate everyday procedures. The Scripting Language is especially useful for writing scripts used by script keepalives for your specific service requirements. For details on script keepalives, refer to the *Cisco Content Services Switch Content Load-Balancing Configuration Guide*.



Note

Commands shown in the script examples are bolded for clarity.

You can write scripts that use any command in the command line interface (CLI). Scripts can also take advantage of logical blocks that run specific commands based upon a set of conditionals and expressions.

This chapter contains the following major sections:

- [Script Considerations](#)
- [Playing a Script](#)
- [Using the Command Scheduler](#)
- [Using the echo Command](#)
- [Using Commented Lines](#)
- [Using Variables](#)
- [Using Logical and Relational Operators and Branch Commands](#)
- [Special Variables](#)
- [Using Arrays](#)
- [Capturing User Input](#)

- [Using Command Line Arguments](#)
- [Using Functions](#)
- [Bitwise Logical Operators](#)
- [Syntax Errors and Script Termination](#)
- [Using the grep Command](#)
- [Using socket Commands](#)
- [Displaying Scripts](#)
- [Script Upgrade Considerations](#)
- [Using the showtech Script](#)
- [Script Keepalive Examples](#)

Script Considerations

When you upgrade the CSS software, the old scripts remain in the old software version's /script directory. For details about copying the scripts from the old software directory to the new software directory, see [“Script Upgrade Considerations”](#) later in this chapter.

The CSS Scripting Language allows you to pass 128 characters in a quoted argument. Assuming an average of seven characters per argument (plus a space delimiter), you can potentially use a maximum of 16 arguments in one script.

Playing a Script

A script executes from within the /script directory on your local hard disk or flash disk. Only scripts that you put in this directory are accessible to the **script play** command.

Use the **script play** command to execute a script line-by-line from the CLI in SuperUser mode. You can also use this command to pass parameter values to a script.

For example, enter:

```
# script play MyScript "Argument1 Argument2"
```

You can display a list of available scripts using the **show script** command.

Using the Command Scheduler

You can schedule the execution of CLI commands to schedule periodic content replication, the gathering of statistics, and scheduled configuration changes. Use the **cmd-sched** command to configure the scheduled execution of any CLI commands, including playing scripts. The commands that will be executed are referred to as the command string. To schedule commands, you must create a configuration record; the record includes the command string and a provision about when to execute the commands.

At the specified time, the command scheduler executes a command string by creating a pseudo-login shell where each string is executed. A **cmd-sched** record is scheduled for execution only upon completion of its shell. Use the **show lines** command to display information about active pseudo-shells.

**Note**

To terminate the execution of a command string, use the **disconnect** command.

The syntax and options for this global configuration mode command are:

- **cmd-sched** - Enable command scheduling.
- **cmd-sched record** *name minute hour day month weekday* “*commands...*” {*logfile_name*} - Create a configuration record for the scheduled execution of any CLI commands, including the playing of scripts.

The variables are listed below. When entering minute, hour, day, month, and weekday variables, you may enter a single integer, a wildcard (*), a list separated by commas, or a range separated by a dash (-).

- *name* - The name of the configuration record. Enter an unquoted text string with a maximum of 16 characters.
- *minutes* - The minute of the hour to execute this command. Enter an integer from 0 to 59.
- *hour* - The hour of the day. Enter an integer from 0 to 23.
- *day* - The day of the month. Enter an integer from 0 to 31.
- *month* - The month of the year. Enter an integer from 1 to 12.
- *weekday* - The day of the week. Enter an integer from 1 to 7. Sunday is 1.

- *command* - The commands you want to execute. Enter a quoted text string with a maximum of 255 characters. Separate multiple commands with a semicolon (;) character. If the command string includes quoted characters, use a single quote character; any single quoted characters not preceded by a backslash (\) character is converted to double quotes when the command string is executed.
- *logfile_name* - An optional variable that defines the name of the log file. Enter a text string with a maximum of 32 characters.

Any of the time variables can contain one or some combination of the following values:

- A single number to define a single or exact value for the specified time variable.
- A wildcard (*) character matching any valid number for the specified time variable.
- A list of numbers separated by commas, with a maximum of 40 characters, to define multiple values for a time variable.
- Two numbers separated by a dash (-) character indicating a range of values for a time variable.

For example:

```
(config)# cmd-sched record periodic_shows 30 21 3 6 1 "show
history;show service;show rule;show system-resources"
```

To enable command scheduling, enter:

```
(config)# cmd-sched
```

To disable command scheduling, enter:

```
(config)# no cmd-sched
```

To delete a configuration record, enter:

```
(config)# no cmd-sched periodic_shows
```

Showing Configured Command Scheduler Records

Use the **show cmd-sched** command to display the state of the command scheduler and information about the records for the scheduled CLI commands. The syntax and options are:

- **show cmd-sched** - Lists the state of the command scheduler and all scheduled CLI command records
- **show cmd-sched name** *record_name* - Lists information about the specified scheduled CLI command record

To view the command scheduler state and all scheduled CLI command records, enter:

```
(config)# show cmd-sched
```

[Table 8-1](#) describes the fields in the **show cmd-sched** command output.

Table 8-1 *Field Descriptions for the show cmd-sched Command*

Field	Description
Cmd Scheduler	State of the command scheduler (enabled or disabled) and the number of configured records.
Sched Rec	The name of the configuration record.
Id	The ID for the record.
Next exec	The date and time the record will be executed.
Executions	How many times the record has executed.
MinList	The configured minute of the hour to execute the command.
HourList	The configured hour of the day to execute the command.
DayList	The configured day of the month to execute the command.
monthList	The configured month of the year to execute the command.
WeekdayList	The configured day of the week to execute the command. Sunday is 1.
Cmd	The commands you want to execute. Separate multiple commands with a ; (semicolon) character.

Using the echo Command

When you play a script, the default behavior of the script is to display each command and its output. Use the **echo** command to control what appears on the screen during script execution. Because the goal of most scripts is to present clear, useful output, it is good practice to disable the **echo** command by using the **no echo** command. The **no echo** command tells the script engine not to print the commands being processed or their output to the terminal. Once you disable the **echo** command, you must use the **echo** command explicitly to print text to the screen.

Echo is enabled automatically upon script termination. For a further explanation of the **echo** and **no echo** commands, see [“Using the “! no echo” Comment”](#) later in this chapter.

**Note**

All of the examples and their outputs shown in the remainder of this chapter assume that the **echo** command is disabled, unless otherwise stated.

Using Commented Lines

To write scripts that other users can understand and maintain, it is important to document your script with comments. Comments are particularly important when other users begin using and/or modifying your scripts. To make this process easier for you, you can add commented lines to your scripts. You denote comments with the exclamation mark (!) as the first character in any line of a script.

For example, enter:

```
! I want a comment here so that I can tell you that I will
! execute a show variables command
show variables
```

This example begins with a comment for the user to read (note the exclamation mark). Any characters are allowed after a comment symbol, but the entire line must be a comment. In the third line, the script will execute a **show variables** command because it does not start with an exclamation mark.

This is an example of a valid statement:

```
! Say hello
echo "Hello"
```

This is an example of an invalid statement:

```
echo "Hello" ! Say hello
```

Using the “! no echo” Comment

To disable the **echo** command without the text “no echo” appearing in the script output, use the commented **no echo** command as the first line in any script. A script actually executes the commented command **no echo**. (If you are familiar with MS-DOS batch files, this command is similar to the **@echo off** DOS command.) For example, enter:

```
! no echo
echo "Hello"
```

The output is:

```
Hello
```

If you enter:

```
! Print Echo
echo "Hello"
```

The output is:

```
echo "Hello"Hello
```

This happens because the script tells the script engine to print the **echo** command to the screen. The result is that the script engine prints the **echo** command and its argument (“Hello”) to the screen, followed by the output (Hello) of the command. This is usually not a desired result, so you typically start most scripts with the **!no echo** command as the first line.

Using Variables

The CLI supports user-defined variables that you can use to construct commands, command aliases, and scripts. Variables are case-sensitive and can be of type integer or character. They can exist as either single elements or arrays of elements. All arrays are of type character, but their elements can be used in integer expressions. For details on arrays, see [“Using Arrays”](#) later in this chapter.

Within a script, you can create and remove variables. During script termination, the script engine automatically removes script-created variables from memory. You can also create a session variable that allows a variable to remain in the session environment after a script has exited. Additionally, if you save a session variable to a user's profile, the variable will be recreated in the CLI session environment upon login. For details on saving a session variable to a user profile, refer to Chapter 3, [Configuring User Profiles](#).

A variable name can contain 1 to 32 characters. Its value is quoted text that normally contains alphanumeric characters. Spaces within the quoted text delineate array elements.

Creating and Setting Variables

To create a variable and set the variable with a value, use the **set** command. For example, enter:

```
set MyVar "1"
```

This command sets the variable MyVar to a value of 1. You can also set the variable in memory without a value. For example, enter:

```
set MyVar ""
```

This will set the variable equal to NULL (no value). This is different from a value of 0, which is a value. You can set the variable so that it can be used across all CLI sessions. For example, enter:

```
set MyVar "1" session
```

Saving a variable marked with the session keyword to your user profile allows you to use the variable across CLI sessions.

To use a variable, you must supply a variable indicator to allow the CLI to recognize the variable. The CLI searches each line of text it processes for the variable indicator “\$”. The next character *must* be the left brace character ({) followed by the variable name. You terminate the variable name with the right brace character (}). For example, if you want to print your variable to the screen using the **echo** command, enter:

```
set MyVar "CSS11506"  
echo "My variable name is: ${MyVar}"
```

The output is:

```
My Variable name is: CSS11506
```

Variable Types

The CLI stores a variable as either type integer or character. The CLI determines a variable's type by the alphanumeric characters in its value. If any non-numeric characters are present, the variable's type is character. If all the characters are numeric, the variable's type is integer.

Arithmetic operations on quoted numbers such as “100” are possible, but are not possible on variables like “CSS11506” because the CLI knows that “CSS11506” is not a numeric value. You can retrieve the variable type by appending [*] at the end of the command string. For example, enter:

```
set MyVar "CSS11506"  
echo "My variable type is ${MyVar}[*]."  
set Number "100"  
echo "My variable type is ${Number}[*]."
```

The output from this script is:

```
My variable type is char.  
My variable type is int.
```

“Int” means integer (numeric with no decimal precision) and “char” means character (any printable ASCII character). Anything that is not an integer is a character. A variable that is defined as “3.14” is a character because the CLI does not consider a period (.) to be a number.

Removing Variables

To remove a variable from memory, use the **no set** command.

For example, enter:

```
no set MyVar
```

If MyVar exists, this line removes the variable from memory. If the variable does not exist, the CLI displays an error message informing you of this invalid action.

**Note**

To permanently remove a session variable, you must also remove it from the user's profile.

Modifying Integer Variables

This section includes the following topics:

- [Using the No Set and Set Commands](#)
- [Using Arithmetic Operators](#)
- [Using the Increment and Decrement Operators](#)

Using the No Set and Set Commands

To modify a variable, use the **no set** command to remove a variable from memory, then use the **set** command with the same variable name to reset the variable with a different value. You can also issue a **set** command on the same variable multiple times without using the **no set** command in between **set** commands.

Example 1:

```
set MyVar "1"  
no set MyVar  
set MyVar "2"
```

Example 2:

```
set MyVar "1"  
set MyVar "2"
```

Example 3:

```
set MyVar1 "1"  
set Myvar2 "2"  
set MyVar1 "${MyVar2}"
```



Note

You can also apply the **set** and **no set** commands to character variables.

Using Arithmetic Operators

To change the value of a variable with arithmetic operators (-, +, /, *, or modulus), use the **modify** command. For example, enter:

```
set MyVar "100"  
modify MyVar "+" "2"  
echo "Variable value is ${MyVar}."  
modify MyVar "-" "12"  
echo "Variable value now is ${MyVar}."  
modify MyVar "*" "6"  
echo "Variable value now is ${MyVar}."  
modify MyVar "/" "6"  
echo "Variable value now is ${MyVar}."  
modify MyVar "MOD" "10"  
echo "Variable modulus value now is ${MyVar}"
```

The output is:

```
Variable value is 102.  
Variable value now is 90.  
Variable value now is 540.  
Variable value now is 90.  
Variable modulus value now is 0.
```

For simple arithmetic operations, the **modify** command takes an operator in quotes (for example, "/", "*", "+", "-", or "MOD") and a new value in quotes. This value does not have to be a constant (for example, "5" or "10"), but can be another variable (for example, "\${Var1}" or "\${Var2}"). The modulus operator "MOD" divides the variable by the specified value (in the above example "10") and sets the variable value to the remainder.

The following sections describe additional operations you can perform on variables using the **modify** command. For more information on the **modify** command, refer to the *Cisco Content Services Switch Command Reference*.

Using the Increment and Decrement Operators

The scripting language provides two operators for incrementing and decrementing variable values. The increment operator “++” adds 1 to the variable value and the decrement operator “--” subtracts 1 from the variable value. Use these operators with the **modify** command.

For example, enter:

```
set MyVar "1"
echo "Variable is set to ${MyVar}."
modify MyVar "++"
echo "Variable is set to ${MyVar}."
modify MyVar "--"
echo "Variable is set to ${MyVar}."
```

The output is:

```
Variable is set to 1.
Variable is set to 2.
Variable is set to 1.
```

These two operators make it possible to add or subtract a value without having to type an addition modification command. So you can replace:

```
modify MyVar "+" 1
```

With:

```
modify MyVar "++"
```

**Note**

Both the increment and the decrement operators work only with integer variables. If you use them with character variables, such as “CSS11506”, an error occurs.

Using Logical and Relational Operators and Branch Commands

To build structured command blocks, use the **if** and **while** branch commands. The **if** command creates script branches based on the results of previous commands. The **while** command is a looping mechanism. Both commands facilitate efficient scripts with fewer repeated actions.

Use both these branch commands with the **endbranch** command, which indicates to the CLI the end of a logical block of commands. Any branches created without a terminating **endbranch** command produce a script logic error and possibly a script syntax error. For information on script errors, see [“Syntax Errors and Script Termination”](#) later in this chapter.

**Note**

You can nest a maximum of 32 levels of branch commands.

Boolean Logic and Relational Operators

You can use each of the following operators with an **if**, **while**, or **modify** command.

The Boolean logic operators used with branch commands are:

- **AND** - Logical AND
- **OR** - Logical OR

The relational operators used with branch commands are:

- **GT** - Greater than
- **LT** - Less than
- **==** - Equal to
- **NEQ** - Not equal to
- **LTEQ** - Less than or equal to
- **GTEQ** - Greater than or equal to

Using the if Branch Command

To create a branch in a script based on the result of a previous command, use the **if** command. If the previous result satisfies the expression in the **if** statement, the script engine executes every line between the **if** and **endbranch** commands. Otherwise, the script engine ignores the intervening commands and execution continues following the **endbranch** statement.

```
set MyVar "1"
if MyVar "==" "1"
    echo "My variable is equal to ${MyVar}!!!"
endbranch
if MyVar "NEQ" "1"
    echo "My variable is not equal to 1 (oh well)."
```

In the example above, the script tests the variable MyVar to see if it is equal to "1". If it is equal to this value, the **echo** command between the **if** command and the **endbranch** command is executed. Note that the variable MyVar does not have the typical variable indicator symbol (\$) in front of it. This is because the **if** command requires that a constant value or a variable name immediately follow the command.

An exception to this rule applies when the **if** command references an array element. In this case, you must use the normal variable syntax, including the variable indicator (\$) and the braces ({ }). For information on arrays, see [“Using Arrays”](#) later in this chapter.

For example, the following logical block is valid:

```
if 12 "==" "${MyVar}"
    echo "We made it!"
endbranch
```

However, this logical block is not valid:

```
if "12" "==" "${MyVar}"
    echo "We made it!"
endbranch
```

Because the **if** command expects to see a constant or a variable name (without the variable indicator), the text string "12" does not satisfy this requirement.

You can also test a variable for a NULL value. For example, enter:

```
if MyVar
    echo "MyVar is equal to ${MyVar}"
endbranch
```

Using the while Branch Command

To execute the same commands repeatedly based on the result of an associated expression, use the **while** branch command. When the expression results in a true value (greater than 1), the script engine executes the commands within the branch. If the result is a false value (denoted by the value of 0), then the script breaks out of the loop and continues execution of all the commands following the **endbranch** command. For example, enter:

```
set Counter "0"
while Counter "NEQ" "5"
    echo "Counter is set to ${Counter}."
    modify Counter "++"
endbranch
echo "We're done!"
```

The output of this logical block is:

```
Counter is set to 0.
Counter is set to 1.
Counter is set to 2.
Counter is set to 3.
Counter is set to 4.
We're done!
```

Until the expression is *not* satisfied, notice that the script jumps to the beginning of the loop and evaluates the expression each time it reaches the **endbranch** command. For the first five times through the loop, Counter is set to 0, 1, 2, 3, and 4, respectively, because the script continually increments the variable. However, on the sixth time through the loop, Counter equals a value of 5, which does not satisfy the expression "While Counter is not equal to 5". The expression produces a false result, which causes the loop to terminate.

As with the **if** command, an **endbranch** command terminates a **while** command logical block.

Special Variables

Within the CLI, there are sets of predefined variables that you can use in scripts to add more power to your scripts. Some of these predefined variables change how scripts react, while others are merely informational variables. None of the special variables requires user intervention for cleanup. However, it is good practice to remove both the `CONTINUE_ON_ERROR` and the `EXIT_MSG` variables following the execution of their relevant command blocks.

Informational Variables

The informational variables are:

- **LINE** - Line name (for example, `pty1`, `console`, etc.).
- **MODE** - Current mode of command (for example, `configure`, `boot`, `service`).
- **USER** - The currently logged in user (for example, `admin`, `bob`, `janet`).
- **ARGS** - A list of arguments passed to a script from the CLI. See [“Using Command Line Arguments”](#) later in this chapter.
- **UGREP** - A line of text obtained using the `grep -u` command.
- **CHECK_STARTUP_ERRORS** - A session variable that determines whether or not a user is informed of startup errors upon login.

CONTINUE_ON_ERROR Variable

Use the `CONTINUE_ON_ERROR` variable to control how a script that is executing in an interactive CLI session handles command failure. By default, a script terminates when it encounters an error. For example, if you use the `echo` command to print out information from a script and spell the command incorrectly, the script exits with a syntax error and prints out the line in which the error occurred. For example, enter:

```
! Spell echo incorrectly
eco "This will not print"
```

The output is:

```
Error in script playback line:2
>>>eco "Hello"
      ^
%% Invalid input detected at '^' marker.
Script Playback cancelled.
```

Because the script contains a spelling error, the script exits with a message telling you what went wrong.

However, there may be cases where you want a script to continue on error. You can override the default behavior by setting the `CONTINUE_ON_ERROR` variable. When you set this variable (regardless of its value), a script continues to execute even after it encounters syntax errors or other errors.

**Note**

Exercise caution when using this variable because the CLI ignores syntax errors when the variable is set. You should set and then unset this variable in scripts where you expect a command to fail.

For example, enter:

```
set CONTINUE_ON_ERROR "1"
! Spell echo incorrectly
eco "This will not print"
echo "This will print"
no set CONTINUE_ON_ERROR
```

The output is:

```
This will print
```

Notice in the above example that the script does not print “Script Playback cancelled” and then terminate. This is because the `CONTINUE_ON_ERROR` variable is set. In most situations, it is important that you issue a **no set** command on the `CONTINUE_ON_ERROR` variable. If you want the script to continue on specific errors, then you have to set the variable and then unset it when you are done. If you do not perform a **no set** command on the variable, then any other syntax errors that occur in the script will not cause an early termination. Remember, setting this variable to a value of 0 does not disable it. To disable the variable’s functionality, you must unset it.

STATUS Variable

Use the STATUS variable to return the exit status of the previously executed CLI command. In most cases, except for the **grep** command, an exit status of 0 indicates that a command was successful, while a non-zero value indicates that a command failed. The CLI sets the STATUS variable automatically after each command completes execution.

**Note**

Using the **grep** command sets the STATUS variable equal to the number of lines that satisfied the search. For details on the **grep** command, see [“Using the grep Command”](#) later in this chapter.

Typically, it is not necessary to examine the STATUS variable because a script will terminate if a command does not execute properly. However, if you set the CONTINUE_ON_ERROR variable, you can use the STATUS variable to test the results of a command.

For example, enter:

```
set CONTINUE_ON_ERROR "1"
echo "Hello world"
if STATUS "NEQ" "0"
    echo "Failure to execute command correctly"
endbranch
```

In the above example, the STATUS variable is set to a non-zero value. This value is specific to the type of error that occurred. In this case, the script receives a general syntax error, which informs you that the command being executed failed. This is a typical example and one that you should watch closely when using the CONTINUE_ON_ERROR variable. In most circumstances, you will want to catch syntax errors as real errors.

**Note**

When writing scripts, keep in mind that the value of the STATUS variable changes as each command executes. If you intend to use a STATUS value later in a script, you should save the value of the STATUS variable in another variable.

In the following example, notice that the error that is most likely to occur is not a syntax error, but an error with the command being initiated. In this case, a nonzero value results in a “Failure to connect to remote host” message. This could be a special case where you want to catch the error and then perform another action instead.

```
set CONTINUE_ON_ERROR "1"
socket connect host 1.1.1.1 port 9
if STATUS "NEQ" "0"
    echo "Failure to connect to remote host"
endbranch
no set CONTINUE_ON_ERROR
```

EXIT_MSG Variable

Use the EXIT_MSG variable to tell the CLI to print out a custom message when a script exits. Typically, you set this variable to a string value that is printed when the script terminates. Set this variable to prepare for potential errors, and unset it using the **no set** command before the script exits cleanly. This variable allows you to take advantage of the CLI's exit upon error behavior, while permitting the flexibility of customizing the error message. For example, enter:

```
set EXIT_MSG "Failure to connect to host"
socket connect host 1.1.1.1 port 9
no set EXIT_MSG
```

The example above shows how you can create a custom error message that will print “Failure to connect to host” if the **socket connect** command returns a non-zero STATUS variable. When this occurs (unless the CONTINUE_ON_ERROR variable is set), the script terminates automatically and the CLI prints the EXIT_MSG string to the screen.

If the **socket connect** command succeeds, then the CLI executes the next command in the script. In this case, the script performs a **no set EXIT_MSG** command. This allows the script to terminate normally without printing an exit message to the screen, which would be inappropriate because no error occurred.

SOCKET Variable

The SOCKET variable contains the connection ID associated with a host. When you connect to a remote host, the SOCKET variable is set so that you can send and receive messages by referring to this variable. When you use the **socket** commands, the SOCKET variable is set automatically (see “Using socket Commands” later in this chapter). When you make multiple connections using the **socket** commands, save the SOCKET variable to another variable or it will be overwritten.

For example, enter:

```
set EXIT_MSG "Failure to connect to host"
socket connect host 1.1.1.1 port 80
no set EXIT_MSG
set EXIT_MSG "Send: Failure"
socket send ${SOCKET} "GET /index.html\n\n"
no set EXIT_MSG
! Save current socket ID
set OLD_SOCKET "${SOCKET}"
! The new socket connect command will overwrite the old
! ${SOCKET} variable
set EXIT_MSG "Failure to connect to host"
socket connect host 1.1.1.1 port 80
no set EXIT_MSG
set EXIT_MSG "Send: Failure"
socket send ${SOCKET} "GET /index.html\n\n"
no set EXIT_MSG
set EXIT_MSG "Waitfor: Failed"
socket waitfor ${OLD_SOCKET} "200 OK"
socket waitfor ${SOCKET} "200 OK"
! Finished, cleanup
no set EXIT_MSG
socket disconnect ${OLD_SOCKET}
socket disconnect ${SOCKET}
```

Using the show variable Command

Use the **show variable** command to display all the variables currently set in the CSS software environment.

The CLI uses the following special variables in its operation to control session behavior and to enhance interaction with CLI commands and the user:

- The **USER** variable is set automatically to the username starting the CLI session at login time.
- The **LINE** variable is set automatically to the line which the user is connected to at login time.
- The **MODE** variable is set automatically to the current mode as the user navigates the hierarchy of CLI modes.
- The **STATUS** variable is set automatically to return the exit status of the previously executed CLI command. In most cases, with the exception of the “grep” command, an exit status of 0 indicates a command was successful, and a non-zero value indicates failure.
- The **CHECK_STARTUP_ERRORS** variable, if set within a profile script, indicates the user should be informed of startup-errors upon login. If the startup-errors file is found in the log directory, the screen displays the “***Startup Errors occurred on boot.***” message.
- The **CONTINUE_ON_ERROR** variable controls how a script executing in an interactive CLI session handles a command error. When you set this variable in a script with the **set** command, the execution of a script continues when errors are detected. If you do not set this variable in a script, the script terminates when an error occurs.

You should exercise caution when using this variable. Syntax errors are ignored when it is set. You should set this variable in the script where you expect a command to fail and then disable it with the **no set** command.

For example, enter:

```
show variable
```

The output is:

```
$MODE = super
$LINE = console
$CHECK_STARTUP_ERRORS = 1 *Session
$UGREP = Weight: 1 Load: 255
$SOCKET = -1
$USER = admin
$STATUS = 0
```

Notice in this example that there are several variables already defined in the environment. You can also display a specific variable by invoking the **show variable** command with a parameter that represents the variable name you want to see.

For example, enter:

```
show variable LINE
```

The output is:

```
$LINE = console
```

Using Arrays

A variable can hold subvalues (elements) within its memory space. Such a variable is commonly called a variable array or just an array. An array can hold numeric values, strings, or both. To create an array, simply create a variable using the **set** command and separate all of the array elements by spaces. For example, enter:

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"
```

You can print the array like any other variable. For example, enter:

```
echo "Days of the week: ${WeekDays}."
```

The output is:

```
Days of the week: Sun Mon Tues Wed Thurs Fri Sat.
```

However, if you want to print out each day separately, you must refer to a single element within the array. For example, enter:

```
echo "The first day of the week is ${WeekDays}[1]."  
echo "The last day of the week is ${WeekDays}[7]."
```

The output is:

```
The first day of the week is Sun.  
The last day of the week is Sat.
```

Notice that you reference the array elements by putting the element number within brackets ([and]) to tell the CLI which element you want to use. You append the brackets to the end of the variable name (including variable indicator and braces).

**Note**

The CSS Scripting Language numbers elements starting at 1, not at 0. Some scripting/programming languages “zero index” their arrays; this scripting system does not.

If you reference an element beyond the boundary of the array, you receive a syntax error. For example, enter:

```
echo "The last day of the week is ${WeekDays}[8]"  
%% Error variable syntax
```

This occurs because there is not an eighth element in the WeekDays array.

Element Numbers

To find out how many elements are in an array, pass the pound symbol (#) rather than an element value. For example, if you want to know how many days are in a week, enter:

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"  
echo "There are ${WeekDays}[#] days in a week."
```

The output is:

```
There are 7 days in a week.
```

You can use this method to figure out how many times to perform a **while** command. For example, enter:

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"
set Counter "1"
while Counter "LTEQ" "${WeekDays} [#]"
    echo "Counter is set to ${Counter}."
    modify Counter "++"
endbranch
```

The output is:

```
Counter is set to 1.
Counter is set to 2.
Counter is set to 3.
Counter is set to 4.
Counter is set to 5.
Counter is set to 6.
Counter is set to 7.
```

Using var-shift to Obtain Array Elements

You may need to print out all the days of the week to the screen. While it is possible to print out each array element by hardcoding the element values, it is not always practical or possible to do this. In this case, it is obvious that there are seven days in the week, but there may be cases where the number of elements in a variable are unknown until the script is executed.

Use the **var-shift** command to push an element out of an array one at a time. For example, enter:

```
set WeekDays "Sun Mon Tues Wed Thurs Fri Sat"
while ${WeekDays} [#] "GT" "0"
    ! Push the 1st element out of the array and shift all
    ! elements up one position.
    echo "Day: ${WeekDays}"
    var-shift WeekDays
endbranch
```

The output of this logical block is:

```
Day: Sun
Day: Mon
Day: Tues
Day: Wed
Day: Thurs
Day: Fri
Day: Sat
```

You cannot use a variable as the index to obtain a given element in an array. So the following statement is invalid:

```
set Pet "Dog Cat"
set Index "1"
echo "First Pet is: ${Pet}[${Index}]"
modify Index "+" 1
echo "Second Pet is: ${Pet}[${Index}]"
```

However, the following statement can solve this problem:

```
set Pet "Dog Cat"
echo "First Pet is: ${Pet}[1]"
var-shift Pet
echo "Second Pet is: ${Pet}[1]"
```

Notice in the second Pet example that there is one less variable needed (you do not need the Index variable) and the first element index (`${Pet}[1]`) is the only one used.

The **var-shift** command deletes the original data within the variable. The variable Pet from the example above now contains only the element "Cat". To solve this problem, save the contents of the original variable in a temporary variable.

For example, enter:

```
set Pet "Dog Cat"
set Temp "${Pet}"
echo "First Pet is: ${Temp}[1]"
var-shift Temp
echo "Second Pet is: ${Temp}[1]"
no set Temp
```

By using the Temp variable, you leave the original variable untouched. If you decide you no longer need the Temp variable, remove it from memory with the **no set** command.

Capturing User Input

To capture user input in a variable, use the **input** command. You can use this command to create a script to assist users in setting up configuration files or preparing a CSS in some predefined way. For example, enter:

```
! Ask the user for his/her full name
echo "What is your full name?"
input FULL_NAME
echo "Hello ${FULL_NAME}!"
```

In the example above, notice that the **input** command has a variable argument called FULL_NAME. Notice that you do not need to set FULL_NAME prior to the **input** command. The command creates the variable and fills it with the data that the user supplies. Also, note that the user's input is terminated by a carriage return.

A user can enter any alphanumeric characters. If a user presses only the Enter key and does not type any characters, then the script creates the variable with a NULL value. This allows you to test the user input to verify that the user typed something. In the following example, the script continues to ask the user the same question until the user types "y".

```
echo -n "\please enter the character 'y' to exit."
input DATA
while DATA "NEQ" "y"
    echo -n "Please enter the character 'y' to exit: "
    input DATA
    echo "\n"
endbranch
```

In the example above, notice the use of the **echo** command with the *-n* parameter. This allows you to prompt the user with a message without forcing a new line at the end of the sentence so that the user's data appears on the same line as the **echo** command output. In the **echo** command quoted string, you can embed "\n", a C-programming style character that puts a line feed in the output to make it more readable.

Using Command Line Arguments

The CLI allows a user to pass command line arguments as quoted text to a script using the **script play** command (see “[Playing a Script](#)”). Use the special reserved ARGV variable to access the command line arguments that a user passed to a script.

To print out all the arguments that a user passed to a script, enter:

```
echo "You passed the arguments: ${ARGV}"
```

If you want to access each argument individually, you can use the ARGV variable as an array, where each argument passed on the command line is separated by spaces. For example, enter:

```
echo "Your first argument passed is: ${ARGV}[1]"
```

The script below (called NameScript) prints a user's first and last names. The script requires that the user pass his/her first and last name (in that order) in quoted text to the script. For example, enter:

```
!no echo
if ${ARGV}[#] "NEQ" "2"
    echo "Usage: NameScript \'First_Name Last_Name\'"
    exit script 1
endbranch
echo "First Name: ${ARGV}[1]"
echo "Last Name: ${ARGV}[2]"
exit script 0
```

This script first tests to see if the user passed any arguments. If the user did not pass exactly two arguments, then the script prints usage information to the screen and exits. If the user passed two arguments, the script assumes that the first argument is the user's first name and the second argument is the user's last name. Finally, the script prints the user's first name and last name to the screen.

For example, to play NameScript, enter:

```
script play NameScript "John Doe"
```

The output is:

```
First Name: John
Last Name: Doe
```

Using Functions

Functions provide a way to organize your scripts into subroutines or modules. You can then call these functions as your script requires them.

To modularize your scripts both for ease of reading and simplification, use the **function** command. This command allows both the creation and calling of script functions. For example, enter:

```
echo "Calling the PrintName function"
function PrintName call
echo "End"
! Function PrintName: Prints the name John Doe
function PrintName begin
echo "My Name is John Doe"
function PrintName end
```

The output is:

```
Calling the PrintName function
My Name is John Doe
End
```

Notice that the command issued between the commands **function PrintName begin** and **function PrintName end** executes before the last **echo** statement in the script. Also note that the script automatically terminates after the last valid line before the function definition.

Passing Arguments to a Function

You can pass a list of arguments to a function. (This is similar to passing command line arguments to a script.) You can also use those arguments when the script calls the function. For example, enter:

```
echo "Calling the PrintName function"
function PrintName call "John Doe"
echo "End"
! Function PrintName: Prints the name John Doe
function PrintName begin
echo "My Name is ${ARGS}"
function PrintName end
```

The output is:

```
Calling the PrintName function
My Name is John Doe
End
```

Notice that the script uses the ARGV variable to hold the passed arguments. Just as command line arguments work with the **script play** command, function arguments work with the **function call** command.

If you pass command line arguments to a script in the **script play** command and use function arguments in the script, then the ARGV variable is stored until the function returns from its execution.

For example, suppose you pass two arguments “Billy Bob” to a script using the **script play** command and the script calls a function called PrintName with different arguments, enter:

```
echo "I was passed the arguments ${ARGV}"
function PrintName call "John Doe"
echo "The original arguments are ${ARGV}"
! Function PrintName: Prints the name John Doe
function PrintName begin
echo "My Name is ${ARGV}"
function PrintName end
```

The output is:

```
I was passed the arguments Billy Bob
My Name is John Doe
The original arguments are Billy Bob
```

Although you use the ARGV variable twice in this script, ARGV had two different values because the function call held its own ARGV variable value independently of the main script’s ARGV variable value.

Now you can modularize your scripts for easier reading and maintenance.

Using the SCRIPT_PLAY Function

To call a script from another script, use the **SCRIPT_PLAY** function. The syntax is:

```
function SCRIPT_PLAY call "ScriptName arg1 arg2..."
```

When you use the **SCRIPT_PLAY** function, use caution when dealing with the **exit script** command. For details on exiting from a script, see [“Exiting a Script Within Another Script”](#) later in this chapter.

Bitwise Logical Operators

The CSS Scripting Language provides two operations for bit manipulation that apply only to numeric values. The bitwise logical operators are:

- **BAND** - Bitwise AND operation
- **BOR** - Bitwise OR operation

You can manipulate the bits of a numeric variable using the **modify** command. For example, if you have a numeric value of 13 and want to find out if the second and fourth bits of the number are turned on (value of 1), you can do so using a bitwise AND operation as follows:

```
          00001101 (13)
AND      00001010 (10)
Result is 00001000 (8)
```

Notice that the fourth bit is common between the values 13 and 10, so the resulting value contains only the common bits. To script this, do the following:

```
set VALUE "13"
modify VALUE "BAND" "10"
echo "Value is ${VALUE}"
```

The output is:

```
Value is 8
```

**Note**

The script overwrites the original value of the variable VALUE. You can use the BOR operator in a similar manner. The mechanics of AND and OR logical operations is beyond the scope of this document.

Syntax Errors and Script Termination

You can create complex scripts using the CSS Scripting Language. During the development of a script, you may encounter a few syntax errors. When a script exits because of a syntax error, the CLI indicates the script line number and the script text where the syntax error occurred.

The CLI also allows you to exit from a script and specify the exit value (zero or non-zero) using the **exit script** command. This lets you know exactly why a script failed. You can then use this information to initiate a decision process to handle the error condition.

Syntax Errors

When you play a script that contains a misspelled command, an unknown command, or a failed command, the script displays a syntax error on the screen. The error message contains the number and the text of the line in the script that caused the failure. The CSS software sets the STATUS variable to the appropriate error code (a non-zero value).

For example, enter:

```
Error in script playback in line: 1
>>>eco "Hello"
      ^
%% Invalid input detected at '^' marker.
Script Playback cancelled.
```

If a script issues a command that returns a non-zero STATUS code, this will also result in a syntax error. For example, if you play a script that issues a **socket connect** command and the host refuses the connection or there is a host name resolution failure, the script terminates with a syntax error.

For example, enter:

```
!no echo
socket connect host 192.168.1.1 port 84 tcp
socket disconnect ${SOCKET}
```

This script works well as long as neither command fails. However, suppose that the host 192.168.1.1 does not exist. Playing the script produces an error as follows:

```
Error in script playback line:2
>>>socket connect host 192.168.1.1 port 84 tcp
CSS11506#
Script Playback cancelled.
```

The script failed because of an error in line 2. Notice that the command is not misspelled and all syntax is correct. However, the command issued a non-zero error code because of its failure to connect. The next command, **socket disconnect**, never executes because of the failure of the first command.

The CLI considers this type of error a “syntax error”. To discover what went wrong, issue the questionable command directly on the CLI.

For example, enter:

```
socket connect host 192.168.1.1 port 84 tcp
%% Failed to connect to remote host
```

The CLI displays the reason why the command failed.



Note

If there are no misspellings in a script, it is good practice to test the commands on the CLI.

Script Exit Codes

When a script terminates, it returns an exit code. The software places the exit code value in the STATUS variable (for reference after the script is invoked). There are two possible script exit code values: zero (success) and non-zero (failure).

To ensure a successful exit code, use the **exit script** command with a value of zero (the default). The integer value is optional with this command.

For example, enter:

```
! Exit Cleanly
exit script 0
```

There may be some cases where you want to indicate that a script failed to run properly. One example of this is when your script requires that a user enter one or more command line arguments. There is no syntax checking that will prove that the user supplied the correct arguments, but you can check if the user supplied the correct number of arguments. For example, enter:

```
if ${ARGS}[#] "NEQ" "2"
    echo "Usage: PingScript \'HostName\'"
    exit script 1
endbranch
```

If this script fails to find exactly two arguments on the command line, it exits with status code 1 (failure). If you were to check the STATUS variable at this point, it would be set to a value of 1.

**Note**

All commands in the CLI write an exit code to the STATUS variable after they execute. If you want to use the STATUS value, you must save it in another variable or use it right away.

For example, enter:

```
script play PingScript
echo "Status: ${STATUS}"
echo "Status: ${STATUS}"
```

The output is:

```
Usage: PingScript "HostName"
Status: 1
Status: 0
```

Because the script contains an **exit script** command with a value of 1, the first **echo** command returns a STATUS value of 1 to indicate that the script failed. The second **echo** command returns a STATUS value of 0 because the first **echo** command executed successfully.

Exiting a Script Within Another Script

It is possible to play a script from another script using the function `SCRIPT_PLAY`. For details on playing a script, see “Using the `SCRIPT_PLAY` Function” earlier in this chapter. In this case, be careful when dealing with the **exit script** command in the secondary script.

If script A invokes script B and script B issue an **exit script** command, both scripts will exit. Therefore, it is important that a script calling another script either removes any **exit script** commands in the second script or makes other arrangements to handle this behavior.

Using the grep Command

To search for specified data and place the last line of the search results in a variable called `UGREP`, use the **grep** command with the `-u` option. For example, to create a script to search for the `Keepalive` field in the **show service** command on a service called `S1`, enter:

```
!no echo
show service S1 | grep -u "Keepalive"
echo "The line is: ${UGREP}"
```

The output is:

```
The line is: Keepalive: (SCRIPT a-kal-pop3 10 3 5)
```

Because the **show service** screen contains the field `Keepalive`, the entire line is stored in the `UGREP` variable. You can also extract each space-separated element by treating the `UGREP` variable as an array. For example, to extract the first block of text, enter:

```
!no echo
show service S1 | grep -u "Keepalive"
echo "The first element in the line is: ${UGREP}[1]"
```

The output is:

```
The first element in the line is: Keepalive:
```

Specifying Line Numbers for Search Results

The search results from a **grep** command can produce multiple lines. In this case, you can specify the line number you want to set in the UGREP variable by issuing **grep -u[n]**, where *[n]* is an optional line number. By default, the last line of the search results is saved in the UGREP variable. Suppose that you want to issue a **grep** command on the **show service** command for the service S1 and search for all the colon (:) characters in the **show service** screen. For example, enter:

```
!no echo
show service S1 | grep -u ":"
echo "The line is: ${UGREP}"
```

The output is:

```
The line is: Weight: 1          Load: 255
```

By default, this is the last line in the **show service** screen that satisfies the search criteria, but it is not the only line that qualifies. To search for a specific line, for example the first line that satisfies the search criteria, use the *[n]* option. For example, enter:

```
!no echo
show service S1 | grep -u1 ":"
echo "The line is: ${UGREP}"
```

The output is:

```
The line is: Name: S1          Index: 1
```

This is the first line that satisfies the search criteria. Notice the **-u1** option, which tells the script to search for the colon character (:) and set the UGREP variable equal to the first qualified search result.

STATUS Results from the grep Command

The STATUS code result from the **grep** command equals the number of qualified search results. This is important to note because other script commands return a zero result to indicate success. The **grep** command returns just the opposite. If it finds 14 matches, then the STATUS variable is set to 14. If it finds no matches, then the STATUS variable is set to 0.

If you search for the string “Keepalive” as illustrated in the previous section, you will find one result. In this case, the STATUS variable is set to 1.

You can combine the status code returned from the **grep** command with a **while** loop to step through all the search results line-by-line.

For example, enter:

```
show service S1 | grep ":"
set endIndex "${STATUS}"
set index "1"
while index "LTEQ" "${endIndex}"
  show service S1 | grep -u${index} ":"
  echo "${UGREP}"
  modify index "++"
endbranch
```

Using socket Commands

To assist you in building a structured protocol, use **socket** commands in a script keepalive. The socket commands allow ASCII or hexadecimal send and receive functionality. Each command that has an optional **raw** keyword converts the data from standard ASCII to hexadecimal. For example, “abcd” is “61626364” in ASCII. In hex, it is “0x61 0x62 0x63 0x64”.

socket connect

To performs either a TCP connection handshake (SYN-SYNACK...) to a specific IP address/port or a UDP connection by reserving the host/port, use the **socket connect** command. The socket value is received in a **SOCKET** variable in the script.

The syntax for this command is:

```
socket connect host ip_address port number [tcp {timeout} {session}
{nowait}]|udp {session}]
```



Note

The software can open a maximum of 64 sockets simultaneously across all scripts on a CSS.

The options and variables are:

- **host** - Keyword that must be followed by the host name or IP address of the remote CSS.
- *ip_address* - The host name or the IP address of the remote CSS.
- **port** - Keyword that must be followed by the port on which to negotiate a connection.
- *number* - The port number on which to negotiate a connection.
- **tcp** - Keyword that specifies a connection using TCP.
- **udp** - Keyword that specifies a connection using UDP.
- *timeout* - Timeout value for making the network connection in milliseconds. If the time limit expires before the connection has been successfully made, then the attempt fails. This applies only to a TCP connection, because UDP is “connectionless”. Enter an integer from 1 to 60000 ms (1 to 60 seconds). The default is 5000 ms (5 seconds).
- **session** - Keyword that tells the socket to remain open until the session ends. If a script opens sockets in the session and does not close them, the sockets remain open until you log out.
- **nowait** - Keyword that tells the socket to send data immediately without waiting to aggregate the data first.

socket send

To write data through a previously connected TCP connection, use the **socket send** command. Note that the **socket send** command clears all currently stored data in the 10-KB receive buffer.

The syntax for this command is:

```
socket send socket_number “string” {raw | base64}
```

- *socket_number* - Socket file descriptor (integer form). This descriptor is returned from the **socket connect** command.
- *string* - Quoted text string with a maximum of 128 characters.

- **raw** -The optional keyword that causes the software to interpret the string values as hexadecimal bytes rather than as a simple string. For example, the software converts “0D0A” to “0x0D 0x0A” (carriage return, line feed).
- **base64** - Encodes the string in the base-64 numbering system before sending it through the connection. This option is useful for HTTP basic authentication when connecting to a password-protected website.

socket receive

To fill up the socket’s 10-KB internal buffer with data from the remote host, use the **socket receive** command. Once the buffer is full, the command locks the buffer so that no new data can be placed in the buffer. You can use the **socket inspect** command to send all data residing in this 10-KB buffer to standard output.



Note

The software removes all previous data from the 10-KB internal buffer before it stores new data.

The syntax for this command is:

```
socket receive socket_number {timeout} {raw}
```

The options and variables are:

- *socket_number* - Socket file descriptor (integer form). This descriptor is returned by the **socket connect** command.
- *socket_number* - Socket file descriptor (integer form). This descriptor is returned by the **socket connect** command.
- *timeout* - The optional timeout value that specifies the number of milliseconds the CSS software waits before the script locks the internal 10-KB buffer and resumes execution. Enter an integer from 1 to 15,000 ms. The default is 100 ms.
- **raw** - The optional keyword that causes the software to interpret the string values as hexadecimal bytes rather than as a simple string. For example, the software converts “0D0A” to “0x0D 0x0A” (carriage return, line feed).

socket waitfor

To fill up the socket's 10-KB internal buffer with data from the remote host, use the **socket waitfor** command. This command is similar to **socket receive** except that it returns immediately upon finding the specified *string* argument. Once the CSS finds the specified string, it returns a $\{\text{STATUS}\}$ value of 0 (success). Otherwise, it returns 1. You can further view the retrieved data using the **socket inspect** command, as described later in this section.

The syntax for this command is:

```
socket waitfor socket_number [anything {timeout}|"string" {timeout}  
  {case-sensitive} {offset bytes} {raw}]
```

The options and variables are:

- *socket_number* - Socket file descriptor (integer form). The descriptor value is returned by the **socket connect** command.
- **anything** - Any incoming data returns the call within the timeout period. If any data is found, the command returns immediately and does not wait the entire timeout period.
- *timeout* - The optional timeout value that specifies the number of milliseconds the CSS waits to find the *string* argument. Enter an integer from 1 to 15000 ms. The default is 100 ms.
- *string* - The specific string that the CSS must find to result in a $\{\text{STATUS}\}$ value of 0. Once the CSS finds the string, the command returns immediately and does not wait the entire timeout period specified by the *integer* argument.
- **case-sensitive** - The optional keyword specifying that the string comparison is case sensitive. For example, "User:" is not equivalent to "user:".
- **offset bytes** - The optional keyword and value indicating the number of bytes after the beginning of the received data to find the string. For example, if you specify a string value of a0 and an offset of 10, then the CSS will look for "a0" 10 bytes after the beginning of the received data.
- **raw** - The optional keyword that causes the software to interpret the string values as hexadecimal bytes rather than as a simple string. For example, the software converts "0D0A" to "0x0D 0x0A" (carriage return, line feed).

socket inspect

To inspect the socket's internal data buffer for actual data, use the **socket inspect** command. If the software finds data, it displays to standard output the last 10 KB of data received. If the displayed characters are non-printable, the software represents them with a period character (.) for readability. (See the example under the *pretty* argument, below.)

The syntax for this command is:

```
socket inspect socket_number {pretty} {raw}
```

The options and variables are:

- *socket_number* - Socket file descriptor (integer form). This descriptor is returned by the **socket connect** command.
- **raw** - Displays the string values as hexadecimal bytes instead of a simple string. For example, instead of printing “ABCD” to standard output, it prints “41424344” (1-byte hexadecimal equivalent).
- **pretty** - Prints each line with both the hexadecimal and the ASCII equivalent for each byte of data. The software prints up to 16 bytes on each line. For example, enter: “0x41 0x42 0x43 0x44 0x10 0x05 ABCD..”



Note

If you use the **socket inspect** command in a keepalive script, you must use the “use-output” option in the command line when configuring the keepalive type for the service.

socket disconnect

To close the connection to the remote host by sending RST (reset) to the remote host so that it knows that the CSS is finished sending data, use the **socket disconnect** command.

The syntax for this command is:

```
socket disconnect socket_number {graceful}
```

- *socket_number* - Socket file descriptor (integer form). This descriptor is returned by the **socket connect** command.

- **graceful** - Closes the connection gracefully by sending a FIN (finished) rather than RST to the remote host.

Socket Administration

You can have a maximum of 64 open (in use) sockets on a CSS concurrently. If you issue a **socket connect** command and do not issue a **socket disconnect** command for that socket's file descriptor (saved in the `#{SOCKET}` variable), then the socket remains open until you issue a **socket disconnect** command with that socket's file descriptor as the value for the `socket_number` argument.

If you open sockets within a script, the sockets close automatically when the script ends, unless you passed the `session` argument in the **socket connect** command. If you open sockets within a session, the sockets close when the session ends (user logs out).

Use the **show sockets** command to list all the socket file descriptors that are currently in use.

[Table 8-2](#) describes the fields in the **show sockets** output.

Table 8-2 *Field Descriptions for the show sockets Command*

Field	Description
Socket ID	Socket file descriptor
Remote Host:Port	The connected Host address and Port pair
Protocol	The protocol specified for the connection: either TCP or UDP.
User	The line identifier as displayed through the show lines command
Time	How long the descriptor has been open



Note

If a remote host times out or closes a socket, the socket architecture cleans up the socket and removes it from the list of used sockets. This cleanup occurs only after you attempt another transfer on a socket that the remote host has already closed. Otherwise, the socket remains idle.

Using the **socket receive** command to retrieve data buffers 10 KB of data at one time. This buffer remains unchanged until you issue another **socket receive** or **socket waitfor** command. At that point, the software clears the buffer and then refills it with more data from the remote host. Each socket descriptor (created by the **socket connect** command) has its own 10-KB buffer.

**Note**

The **socket send** command also clears all currently stored data in the 10-KB buffer. For details, see the “[socket send](#)” section.

Displaying Scripts

To display a list of the scripts that reside in the CSS script directory or the contents of a specific script (with or without line numbers), use the **show script** command. This command is available in SuperUser mode and all configuration modes. The syntax of this command is:

```
show script {filename {line-numbers}}
```

The variables and option of this command are:

- *filename* - Name of a valid script file whose contents you want to display. Enter a case-sensitive unquoted text string with a maximum of 32 characters.
- **line-numbers** - Displays line numbers for each line in the script.

For example, to display a list of all scripts in the CSS script directory, enter:

```
# show script
```

To display the text of the ap-kal-dns keepalive script, including line numbers, enter:

```
# show script ap-kal-dns line-numbers
```

Script Upgrade Considerations

When you upgrade to a new version of CSS software, all script files that you have modified in the previous software version's /script directory need to be copied to the new software version's /script directory or else the scripts remain in the old /script directory and the CSS will not find them.

Follow these steps *before* upgrading your CSS software:

1. Use the **archive script** command in SuperUser mode to archive each script file. For details on archiving a script, refer to Chapter 1, [Managing the CSS Software](#).
2. Upgrade your CSS software. For details on upgrading the CSS software version, refer to Appendix A, [Upgrading Your CSS Software](#).
3. Use the **restore script** command in SuperUser mode to restore the scripts to the /script subdirectory of the new software version. For details for restoring a script, refer to Chapter 1, [Managing the CSS Software](#).

Using the showtech Script

To gather information designed to assist the Cisco Technical Assistance Center (TAC) in analyzing your CSS, use the **showtech** script. The output of the script displays a set of CSS status and configurations settings that the TAC can use for problem resolution. Use the output-capturing capabilities of your application connected to the CSS to save the script output for analysis.

You can also save the script output to the file log/showtech.out by entering **y** at the prompt as shown below. You can then copy the output file and send it to the TAC if necessary.

To run the script, enter:

```
# script play showtech
```

```
Save output to disk [y/n]? y
Output will be saved to log/showtech.out
Please wait...
```

If you enter **n**, then the output appears on your screen.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: showtech
```

```

!
! Description:
!           show tech-support script
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

set CONTINUE_ON_ERROR "1"

no terminal more
llama

show clock
show disk
show running-config
show statistics
show service-internal
show service summary
show service
show system-resources
show dump-status
show core
show circuit all
show arp
show ip route
show phy
show summary
show rule
show group
show ether-errors
show keepalive
show ip stat
show rmon
show bridge status
show bridge forwarding
show interface
show virtual-routers
show critical-services
show redundancy
show chassis inventory
show chassis verbose
show log sys.log tail 200
exit
terminal more

set CONTINUE_ON_ERROR "0"

exit script 0

```

Script Keepalive Examples

The CSS provides scripted keepalives to support the need for keepalives operations that cannot be handled using non-scripted keepalives. We recommend that you limit I/O operations in a scripted keepalive to socket operations used to probe network connectivity to a server and for determining application health on a server. Although the scripting language supports file I/O on the CSS hard drive or flash drive, we recommend that you do not use file I/O operations within scripted keepalives. Extensive file I/O operations within scripted keepalives may cause services to transition. File system access is allowed in scripts executed from the CLI or from the command scheduler.

The following sections provide examples of script keepalives. You can use them as is or modify them for your applications.

Example of a Custom TCP Script Keepalive with Graceful Socket Close (FIN)

Use the following script keepalive to open and gracefully close (using a FIN rather than a RST) a socket on user-specified TCP ports.



Note

The service mode **keepalive tcp-close fin** command or global keepalive mode **tcp-close fin** command also gracefully closes (using a FIN rather than a RST) a socket.

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-tcp-ports
! Parameters: Service Address, TCP Port(s)
!
!Description:

! This script will open and close a socket on the user specified
! ports.
! The close will be a FIN rather than a RST. If one of the ports fails
! the service will be declared down
!
! Failure Upon:
! Not establishing a connection with the host on one of the specified
! ports.
```

Script Keepalive Examples

```

!
! Notes: Does not use output
!       Will handle out of sockets scenario.
!
! Tested: KGS 12/18/01
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

set OUT-OF-SOCKETS "785"
set NO-CONNECT "774"

! Make sure the user has a qualified number of arguments
if ${ARGS} [#] "LT" "2"
    echo "Usage: ap-kal-tcp-ports \ipAddress tcpPort1 [tcpPort2
tcpPort3...]\'"
    exit script 1
endbranch

set SERVICE "${ARGS}[1]"
!echo "SERVICE = ${ARGS}[1]"
var-shift ARGS

while ${ARGS} [#] "GT" "0"
    set TCP-PORT "${ARGS}[1]"
    var-shift ARGS
    function SOCKET_CONNECT call
! If we're out of sockets, exit and look for sockets on the next KAL
interval
    if RETURN "==" "${OUT-OF-SOCKETS}"
        set EXIT_MSG "Exceeded number of available sockets, skipping until
next interval."
        exit script 0
    endbranch

! Valid connection, look to see if it was good
    if RETURN "==" "${NO-CONNECT}"
        set EXIT_MSG "Connect: Failed to connect to
${SERVICE}:${TCP-PORT}"
        exit script 1
    endbranch
endbranch

no set EXIT_MSG
exit script 0

function SOCKET_CONNECT begin
    set CONTINUE_ON_ERROR "1"
    socket connect host ${SERVICE} port ${TCP-PORT} tcp 2000

```

```
set SOCKET-STAT "${STATUS}"
set CONTINUE_ON_ERROR "0"
socket disconnect ${SOCKET} graceful
function SOCKET_CONNECT return "${SOCKET-STAT}"
function SOCKET_CONNECT end
```

Default Script Keepalives

The script keepalives listed below are included in the /script directory of your CSS and defined in the sections following the list:

- [SMTP KEEPALIVE](#)
- [NetBIOS Name Query \(Microsoft Networking\)](#)
- [HTTP List Keepalive](#)
- [POP3 Keepalive](#)
- [IMAP4 Keepalive](#)
- [Pinglist Keepalive](#)
- [Finger Keepalive](#)
- [Time Keepalive](#)
- [Setcookie Keepalive](#)
- [HTTP Authentication Keepalive](#)
- [DNS Keepalive](#)
- [Echo Keepalive](#)
- [HTTP Host Tag Keepalive](#)
- [Mailhost Keepalive](#)
- [LDAP Keepalive](#)

SMTP KEEPALIVE

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-smtp
! Parameters: HostName
!
! Description:
! This script will log into an SMTP server and send a 'hello'
! to make sure the SMTP server is stable and active.
!
! Failure Upon:
! 1. Not establishing a connection with the host.
! 2. Failure to get a good status code after saying 'hello'
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-smtp \'Hostname\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 25 tcp

set EXIT_MSG "Waitfor: Failed"
! Receive the incoming status code 220 "welcome message"
socket waitfor ${SOCKET} "220" 200

set EXIT_MSG "Send: Failed"
! Send the helo to the server
socket send ${SOCKET} "helo ${HostName}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for status code "250" to be returned
socket waitfor ${SOCKET} "250" 200

! We've successfully logged in, the server is up and running.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0

```

NetBIOS Name Query (Microsoft Networking)

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-netbios
! Parameters: Hostname
!
! Description:
! We will make a netbios name query that we know will be
! a "negative" response. RFC-1002 NETBIOS states that a hex
! value of:
! 0x81 Session Request
! 0x82 Positive Session Response
! 0x83 Negative Session Response
! We will key off of 0x83 which states we failed, but which
! also means that the service was stable enough to know that
! we are not a valid machine on the network.
! This script will send an encoded message for Session Request
! (0x81) and will invent a CALLER and a CALLED machine name
! (Caller = this script and CALLED = Server)
!
! Failure Upon:
! 1. Not establishing a connection with the host.
! 2. Not receiving a status code 0x83 (negative response)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-pop3 \'Hostname\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

! Connect to the remote host (default timeout)
set EXIT_MSG "Connection failure"
socket connect host ${HostName} port 139 tcp

! Send a Netbios Session Request (0x81) and its required encoded
! values.
! This value will be sent in RAW Hex
set EXIT_MSG "Send: Failure"
socket send ${SOCKET}
810000442045454550454d454d464a434143414341434143414341434143414341
434100" raw

! Wait for a response code of 0x83
set EXIT_MSG "Waitfor: Failure"
socket waitfor ${SOCKET} "83" raw

```

```
no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0
```

HTTP List Keepalive

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-httpplist
! Parameters: Site1 WebPage1 Site2 WebPage2 [...]
!
! Description:
! This script will connect a list of sites/webpage pairs. The
! user must simply supply the site, and then the webpage and
! we'll attempt to do an HTTP HEAD on that page.
!
! Failure Upon:
! 1. Not establishing a connection with the host.
! 2. Not receiving a status code 200 on the HEAD request on any
! one site. If one fails, the script fails.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "LT" "2"
    echo "Usage: ap-kal-httpplist \'WebSite1 WebPage1 WebSite2 WebPage2
    ...'"
    exit script 1
endbranch

while ${ARGS}[#] "GT" "0"
    set Site "${ARGS}[1]"
    var-shift ARGS

    if ${ARGS}[#] "==" "0"
        set EXIT_MSG "Parameter mismatch: hostname present but webpage
was not"

        exit script 1
    endbranch
    set Page "${ARGS}[1]"
    var-shift ARGS
    no set EXIT_MSG
    function HeadUrl call "${Site} ${Page}"
endbranch
exit script 0
function HeadUrl begin
```

```

echo "Getting ${ARGS}[1] from Site ${ARGS}[2]\n"
! Connect to the remote Host
set EXIT_MSG "Connect: Failed to connect to ${ARGS}[1]"
socket connect host ${ARGS}[1] port 80 tcp

! Send the head request
set EXIT_MSG "Send: Failed to send to ${ARGS}[1]"
socket send ${SOCKET} "HEAD ${ARGS}[2] HTTP/1.0\n\n"

! Wait for the status code 200 to be given to us
set EXIT_MSG "Waitfor: Failed to wait for '200' on ${ARGS}[1]"
socket waitfor ${SOCKET} " 200 "

no set EXIT_MSG
socket disconnect ${SOCKET}

function HeadUrl end

```

POP3 Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-pop3
! Parameters: HostName UserName Password
!
! Description:
!   This script will connect to a POP3 server and login with the
!   username/password pair specified as argument 2 and 3. After which
!   it will log out and return.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to log in with supplied username/password.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-pop3 \'Hostname UserName Password\'"
    exit script 1
endbranch
! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"
set Password "${ARGS}[3]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 110 tcp

```

Script Keepalive Examples

```

set EXIT_MSG "Waitfor: Failed"
! Wait for the OK welcome message for 200ms
socket waitfor ${SOCKET} "+OK" 200

set EXIT_MSG "Send: Failed"
! Send the username to the host
socket send ${SOCKET} "USER ${UserName}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200

set EXIT_MSG "Send: Failed"
! Send the password
socket send ${SOCKET} "PASS ${Password}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0

```

IMAP4 Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-imap4
! Parameters: HostName UserName Password
!
! Description:
!   This script will connect to a IMAP4 server and login with the
!   username/password pair specified as argument 2 and 3. After which
!   it will log out and return.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to log in with supplied username/password.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-imap4 \'Hostname UserName Password\'"
    exit script 1

```

```
endbranch

! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"
set Password "${ARGS}[3]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 143 tcp

set EXIT_MSG "Waitfor: Failed"
! Wait for the OK welcome message for 600ms
socket waitfor ${SOCKET} "OK" 600

set EXIT_MSG "Send: Failed"
! Send the username to the host
socket send ${SOCKET} "a1 LOGIN ${UserName} ${Password}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "a1 OK" 200

set EXIT_MSG "Send: Failed"
! Send the password
socket send ${SOCKET} "a2 LOGOUT\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "a2 OK" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0
```

Pinglist Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-pinglist
! Parameters: HostName1 HostName2 HostName3, etc.
!
! Description:
! This script is designed to ping a list of hosts that the user
! passes in on the command line.
!
! Failure Upon:
! 1. Not being able to ping any one of the hosts in the list
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "LT" "1"
    echo "Usage: ap-kal-pinglist \'HostName1 HostName2 HostName3
    ...\'"
    exit script 1
endbranch

while ${ARGS}[#] "GT" "0"
    set Host "${ARGS}[1]"
    var-shift ARGS
    function PingHost call "${Host}"
endbranch

no set EXIT_MSG
exit script 0

function PingHost begin

! Ping the first host
ping ${ARGS}[1] | grep -u Success
if STATUS "NEQ" "0"
    show variable UGREG | grep 100
    if STATUS "==" "0"
        set EXIT_MSG "Ping: Failure to ping ${ARGS}[1]"
        exit script 1
    endbranch
endbranch

function PingHost end

```

Finger Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-finger
! Parameters: HostName UserName
!
! Description:
!   This script will connect to the finger server on the remote
!   host. It will query for the UserName and receive the
!   information back.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to send/receive data to the host
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "2"
    echo "Usage: ap-kal-finger \'Hostname UserName\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 79 tcp

set EXIT_MSG "Send: Failed"
! Send the username to "finger"
socket send ${SOCKET} "${UserName}\n"
set EXIT_MSG "Waitfor: Failed"
! Wait for data for 100ms (default)
socket waitfor ${SOCKET} "${UserName}"

no set EXIT_MSG
! If the data came in, then we are good to quit
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0

```

Time Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-time
! Parameters: HostName
!
! Description:
!   This script will connect to a remote host 'time' service on
!   port 37 and get the current time. This script currently works
!   strictly with TCP. [Ref. RFC-868]
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to receive incoming data
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-time \'Hostname\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 37 tcp 2000

set EXIT_MSG "Receive: Failed"
! waitfor any data for 2000ms
socket waitfor ${SOCKET} anything 2000

! If the data came in, then we are good to quit
socket disconnect ${SOCKET}

no set EXIT_MSG

exit script 0

```

Setcookie Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-setcookie
! Parameters: HostName WebPage cookieString
!
! Description:
!   This script will keepalive a WWW server that is setting
!   cookies in the HTTP response header.  The header value
!   looks like this:
!   Set-Cookie: NAME=VALUE
!
!   The user will be responsible for sending us the name & value
!   in a string like "mycookie=myvalue" so that we can compare
!   the incoming Set-Cookie: request.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not being able to receive the cookie
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-setcookie \HostName WebPage cookieString\"
    echo "(Where cookieString is a name=value pair like
\'mycookie=myvalue\')"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set WebPage "${ARGS}[2]"
set CookieData "${ARGS}[3]"

! Connect to the remote host (use default timeout)
set EXIT_MSG "Connection Failed"
socket connect host ${HostName} port 80 tcp

! send our request to the host
set EXIT_MSG "Send: Failure"
socket send ${SOCKET} "GET ${WebPage} HTTP/1.0\n\n"

! Wait for the cookie to come in
set EXIT_MSG "Waitfor: Failure"
socket waitfor ${SOCKET} "${CookieData}"

```

```
! Done
no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0
```

HTTP Authentication Keepalive

```
!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-httpauth
! Parameters: HostName WebPage Username-Password
!
! Description:
! This will keepalive an authentication connection by building
! a get request with the Authentication field filled with the
! Username-Password string formatted like so: "bob:mypassword"
! This is critical to make the authentication base64 hash work
! correctly.
!
! Note: This script authentication is based on HTTP AUTHENTICATION
! RFC-2617. Currently only supported option is "Basic"
! authentication using base64 encoding. "Digest" Access is
! not supported at this time.
!
! Failure Upon:
! 1. Not establishing a connection with the host.
! 2. Not being able authenticated with the Username-Password
! (not being given a status code of "200 OK"
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS} [#] "NEQ" "3"
    echo "Usage: ap-kal-httpauth \'Hostname WebPage
Username:Password\'"
    echo "(Ie. ap-kal-httpauth \'192.168.1.1 /index.html
bob:mypassword\')"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set WebPage "${ARGS}[2]"
set UserPass "${ARGS}[3]"

! Connect to the remote Host
set EXIT_MSG "Connection Failure"
socket connect host ${HostName} port 80 tcp
```

```

! Send the GET request for the web page, along with the authorization
! This builds a header block like so:
!
! GET /index.html HTTP/1.0\r\n
! Authorization: Basic bGFiOmxhYnRlc3Qx\r\n\r\n

set EXIT_MSG "Send: Failed"
socket send ${SOCKET} "GET ${WebPage} HTTP/1.0\n"
socket send ${SOCKET} "Authorization: Basic "
socket send ${SOCKET} "${UserPass}" base64
socket send ${SOCKET} "\n\n"

! Wait for a good status code
set EXIT_MSG "Waitfor: Failed"
socket waitfor ${SOCKET} "200 OK"

no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0

```

DNS Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-dns
! Parameters: Server DomainName
!
! Description:
! This script will resolve a domain name from a specific DNS
! server. This builds a UDP packet based on RFC-1035
!
! Failure Upon:
! 1. Not resolving the hosts's IP from the domain name
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-finger \'Hostname\'"
    exit script 1
endbranch

set HostName "${ARGS}[1]

! Connect to the remote host
set EXIT_MSG "Connection failed"
socket connect host ${HostName} port 53 udp

```

```

! This may require a little explanation.  Since we just want to see
! if the DNS server is alive we will send a simple DNS Query.  This
! query is hard coded in hexadecimal and sent raw to the DNS server.
! The DNS request has a 12 byte header (as seen for the first 12 bytes
! of hex) and then a DNS name (ie. www.cisco.com).  Lastly it follows
! with some null termination and a few bytes representing query type.
! See RFC-1035 for more.
set EXIT_MSG "Send: failure"
socket send ${SOCKET}
"0002010000010000000000000377777705636973636f03636f6d0000010001" raw

! Receive some unexplained response.  We don't care what it is because
! an unstable DNS server or a non-existent one would probably not send
! us any data back at all.

set EXIT_MSG "Receive: Failed to receive data"
socket receive ${SOCKET}

no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0

```

Echo Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-echo
! Parameters: HostName [ udp | tcp ]
!
! Description:
!   This script will send a TCP or UDP echo (depending on what the
!   user has passed to us) that will echo "Hello Cisco" to the
!   remote host, and expect it to come back.  The default protocol
!   is TCP.
!
! Failure Upon:
!   1. Not establishing a connection with the host (TCP Only).
!   2. Not being able to retrieve an echoed message back
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "2"
    if ${ARGS}[#] "NEQ" "1"
        echo "Usage: ap-kal-echo \'Hostname [ udp | tcp ]\'"
        exit script 1
    endbranch
endbranch

```

```

! Defines:
set HostName "${ARGS}[1]"
set nProtocol "tcp"

! See if the user has specified a protocol
if ${ARGS}[#] "==" "2"
    ! The user specified a protocol, so reset the value
    set nProtocol "${ARGS}[2]"
endbranch

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 7 ${nProtocol}

set EXIT_MSG "Send: Failed"
! Send the text to echo...
socket send ${SOCKET} "Hello Cisco!\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for the reply from the echo (should be the same)
socket waitfor ${SOCKET} "Hello Cisco!" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.

socket disconnect ${SOCKET}
no set EXIT_MSG
exit script 0

```

HTTP Host Tag Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-httpitag
! Parameters: HostName WebPage HostTag
!
! Description:
!   This script will connect to the remote host and do an HTTP
!   GET method upon the web page that the user has asked for.
!   This script also adds a host tag to the GET request.
!
! Failure Upon:
!   1. Not establishing a connection with the host.
!   2. Not receiving an HTTP status "200 OK"
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

```

if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-httptag \'Hostname WebPage HostTag\'"
    exit script 1
endbranch
! Defines:
set HostName "${ARGS}[1]"
set WebPage "${ARGS}[2]"
set HostTag "${ARGS}[3]"

! Connect to the remote Host
set EXIT_MSG "Connection Failure"
socket connect host ${HostName} port 80 tcp

! Send the GET request for the web page
set EXIT_MSG "Send: Failed"
socket send ${SOCKET} "GET ${WebPage} HTTP/1.0\nHost: ${HostTag}\n\n"

! Wait for a good status code
set EXIT_MSG "Waitfor: Failed"
socket waitfor ${SOCKET} "200 OK"

no set EXIT_MSG
socket disconnect ${SOCKET}
exit script 0

```

Mailhost Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-mailhost
! Parameters: HostName UserName Password
!
! Description:
!   This script will check the status on a mailhost. The mailhost
!   should be running a POP3 and SMTP service. We will attempt
!   to keepalive both services, and if one goes down we will report
!   an error.
!
! Failure Upon:
!   1. Not establishing a connection with the host running an SMTP
!      service.
!   2. Not establishing a connection with the host running a POP3
!      service.
!   3. Failure to get a good status code after saying 'hello' to SMTP.
!   4. Failure to login using POP3.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

```
if ${ARGS}[#] "NEQ" "3"
    echo "Usage: ap-kal-pop3 \'Hostname UserName Password\'"
    echo "(For checking an SMTP and POP3 service)"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"
set UserName "${ARGS}[2]"
set Password "${ARGS}[3]"

!!!! SMTP !!!!!

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 25 tcp

set EXIT_MSG "Waitfor: Failed"
! Receive the incoming status code 220 "welcome message"
socket waitfor ${SOCKET} "220" 200

set EXIT_MSG "Send: Failed"
! Send the hello to the server
socket send ${SOCKET} "helo ${HostName}\n"
set EXIT_MSG "Waitfor: Failed"
! Wait for status code "250" to be returned
socket waitfor ${SOCKET} "250" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

!!!! POP3 !!!!!

set EXIT_MSG "Connection Failed"
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 110 tcp

set EXIT_MSG "Waitfor: Failed"
! Wait for the OK welcome message for 200ms
socket waitfor ${SOCKET} "+OK" 200

set EXIT_MSG "Send: Failed"
! Send the username to the host
socket send ${SOCKET} "USER ${UserName}\n"
```

```

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200
set EXIT_MSG "Send: Failed"
! Send the password
socket send ${SOCKET} "PASS ${Password}\n"

set EXIT_MSG "Waitfor: Failed"
! Wait for confirmation
socket waitfor ${SOCKET} "+OK" 200

! We've successfully logged in, the server is up and going.
! The job was done successfully.
socket disconnect ${SOCKET}

no set EXIT_MSG
exit script 0

```

LDAP Keepalive

```

!no echo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Filename: ap-kal-ldap
! Parameters: HostName
!
! Description:      "Lightweight Directory Access Protocol v3"
! This script will connect to an LDAP server and attempt to
! "bind request" to the server. Once the server gives a
! positive response we will disconnect (RFC-2251).
!
! Bind Response Code we will search for is: 0x0a 0x01 0x00
!
! Failure Upon:
! 1. Not establishing a connection with the host.
! 2. Failure to receive the above response code.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Make sure the user has a qualified number of arguments
if ${ARGS}[#] "NEQ" "1"
    echo "Usage: ap-kal-ldap \'Hostname\'"
    exit script 1
endbranch

! Defines:
set HostName "${ARGS}[1]"

set EXIT_MSG "Connection Failed"

```

```
! Connect to the remote host (use default timeout)
socket connect host ${HostName} port 389 tcp 2000

set EXIT_MSG "Send: Failure"
! Send a Bind Request to the remote host. This is simply a standard !
! "capture" of a bind request in hex. This should work for all standard
! version 3 LDAP servers. socket send ${SOCKET}
"300c020102600702010204008000" raw

set EXIT_MSG "Receive: Failure"
! Expect to receive a standard response from the host. This should !
! be equal to a SUCCESS response code: socket waitfor ${SOCKET} "0a0100"
2000 raw

set EXIT_MSG "Send: Failure"
! Send an exit "Unbind Request" to the remote host so that they ! are
! not left hanging. socket send ${SOCKET} "30050201034200" raw

no set EXIT_MSG
socket disconnect ${SOCKET}

exit script 0
```

■ Script Keepalive Examples