Infoblox
CONTROL YOUR NETWORK

Infoblox CCS Scripting Guide

for NetMRI 6.x and 7.x

Copyright Statements

Trademark Statements

Company Information

Web:http://www.infoblox.com/company/overview/contact

Document Updated: December 29, 2017

Warranty Information

Your purchase includes a 90-day software warranty and a one year limited warranty on the Infoblox appliance, plus an Infoblox Warranty Support Plan and Technical Support. For more information about Infoblox Warranty information, refer to the Infoblox Web site, or contact Infoblox Technical Support.

Product Information

Hardware Models

NetMRI: NetMRI-1102-A, NT-1400, NT-2200, and NT-4000
Infoblox Advanced Appliances: PT-1400, PT-2200, PT-4000, and PT-4000-10GE
Network Insight Appliances: ND-800, ND-1400, ND-2200, and ND-4000
Trinzic Appliances: TE-100, TE-810, TE-820, TE-1410, TE-1420, TE-2210, TE-2220, Infoblox-4010, and Infoblox-4020
All Trinzic Rev-1 and Rev-2 appliances
Cloud NetMRI: CP-V800, CP-V1400, and CP-V2200
Trinzic Reporting: TR-800, TR-1400, TR-2200, and TR-4000
DNS Cache Acceleration Appliances: IB-4030 and IB-4030-10GE

Document Number: 400-0713-000 Rev. A

Infoblox
CONTROL YOUR NETWORK

Job Automation with CCS Scripting

NetMRI's Job Management feature set (**Configuration Management –› Job Management** tab) enables automation of processes for monitoring, network analysis and routine maintenance. Job Management is also the enabling feature in the Automation Change Manager (ACM) system. Job Management and Job Automation operations involve the following tools:

- **Scripts**, used for automation tasks on numerous devices across the network, to execute sequences of Command Line Interface commands on network devices. Scripts can be written in Infoblox's proprietary CCS language or in Perl (using the standard Perl API).

Note: This document focuses on use of the CCS language. For more information on use of the Perl language in job automation, see the *Infoblox NetMRI Administrator Guide*.

- **Jobs**, which are scheduled instances of CCS or Perl scripts that run against selected devices or device groups. End-user credentials can be used for specific jobs;
- **Config Templates**, an easy editing environment in NetMRI to create standard configuration files and rapidly duplicate configurations for deployment;
- **Lists**, tables of data referred to by scripts, for matching purposes during script execution;
- **Custom issues**, which can be raised by CCS or Perl scripts to announce/point out conditions discovered during script processing.

Job automation ensures consistency across devices in the managed network and can save valuable time. Because NetMRI supports both Perl and its own proprietary CCS scripting language, users can employ change automation to do the following:

- Define generic configuration templates for large collections of like devices such as Cisco and Juniper routers and switches;
- Execute mass change rollouts through the downloading of template files, reducing the need to execute sequences of CLI commands and enabling larger-scale changes across the network;
- Reference external lists to populate variables when executing actions against devices.

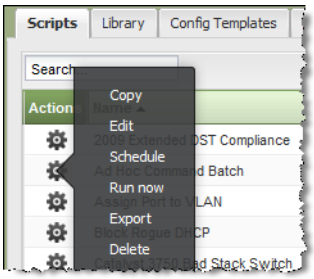
About CCS

Note: NetMRI can only execute scripts on devices for which it has login credentials, even if the devices are within the set specified by Script-Groups and Script-Devices.

CCS is a proprietary scripting language for executing sequences of command-line interface (CLI) commands on NetMRI-supported network devices, to perform job automation tasks. If you know Cisco IOS, writing CCS scripts is relatively straightforward. This document presumes prior knowledge of Cisco IOS and any other CLI interface for devices such as Extreme, Juniper and other vendors supported by NetMRI. Some previous structured programming knowledge will also be helpful.

While CCS is not as powerful or flexible as Perl, the learning curve is not as steep. CCS allows you to quickly develop useful jobs that can run across hundreds of devices in the managed network.

Tools for Using CCS



NetMRI provides its own script editor, which can be found in the (**Configuration Management** –› **Job Management** tab –› **Scripts** page by selecting the Action icon for a script and choosing **Edit**.

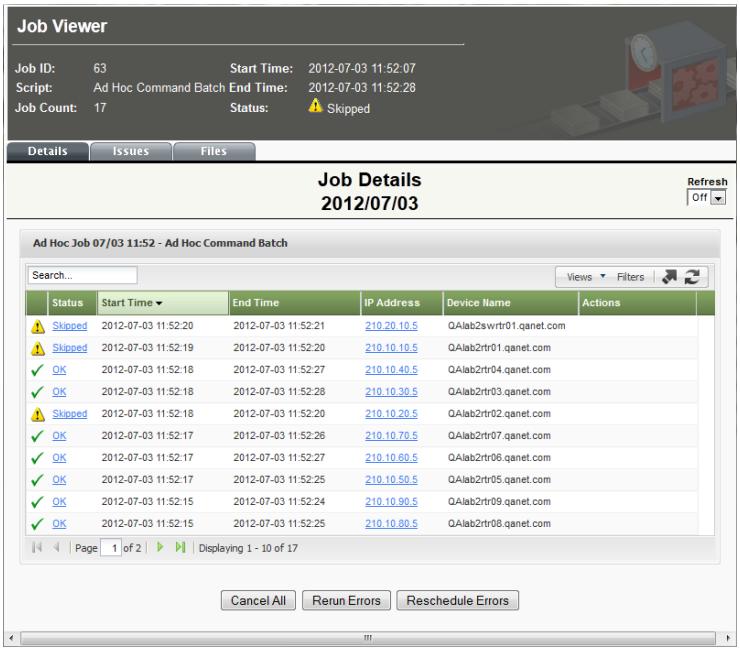
The Edit Script window is the primary NetMRI tool for script editing, but you are not limited to it. A standard text editor such as Notepad, Notepad++ or Notepad 2 may be used to write scripts. To see an indication of how to do so, open the Edit Script window and click **Export**. In the Windows file requester, you can select a preferred text editor to automatically open the script file.

Once you write the script, execute it by selecting **Run Now** from the Action menu. You may run the script against any device group, or select one or more devices in the **Script Run Now** wizard. We advise strictly limiting the number of devices to run a script against until the script is verified to work without problems.

After the script runs, NetMRI automatically displays the Job History page with your current job at the top of the table.

Note: If any devices in the selected Device Group are skipped over during script execution, the Job History will show the job status as Skipped. This does not indicate the script failed to run; it simply indicates that the job did not execute on one or more devices in the job run. This is expected behavior.

To view the job status, click the link in the **Name** column. The Job Viewer appears in a separate browser window, showing the **Details** page. **Details** lists all the devices participating in the job.

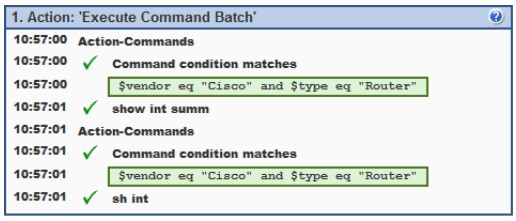


The **Status** column shows the broad result of the script's execution of each device. It does not indicate whether the script's results are what was desired during its run; a status of **OK** means that the script ran to completion on the device.

When a **Skipped** status appears, it indicates a device in the chosen Device Group has been skipped over during the script's execution, most likely due to it being filtered out by a *Script-Filter:* attribute, which is discussed later in this document.

The **Files** tab provides access to any individual text output files defined in the CCS script. CCS uses specific command directives to send desired output from devices to an external text file. Each device may have its own external text file associated with it, and a script may append text multiple times to the same file.

Clicking on the **Status** link (which will read *Skipped* or *OK* in most cases) for any device in the Job Viewer, opens the Job Details Viewer in another browser window. The Job Details Viewer shows the results of the job run on the chosen device.



After a script executes and you click a **Status** link, the Job Details Viewer displays the **Process Log** page. The Process Log shows, in graphic form, the execution sequence for all CCS sections, CCS attributes and automated CLI commands (see *CCS Script Hierarchies* for more information).

The process log highlights all successive matches for the script iteration that ran on the current device. Any successful pattern match against device command output in a script execution appears in green as shown here. A process log may be much longer than the illustrated example. (We return to a more-complete example later in this Guide.)

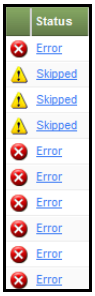
Along with the graphical **Process Log**, the **Status Log** page provides the raw-text version of the same information graphically shown in the **Process Log**, for all statements executed in the script, and their result (whether they matched or were not matched in the script):

```
+++ 1. Action: Execute Command Batch
+++ 1. [Action-Commands]
+++ 1. Command condition ..................................................... MATCH
+++ 1. Sending 'show int summ' ............................................... OK
+++ 1. [Action-Commands]
+++ 1. Command condition ..................................................... MATCH
+++ 1. Sending 'sh int' ...................................................... OK
+++ 1.1. Trigger: Show Ethernet Interfaces
+++ 1.1. Trigger-Template ................................................... MATCH
```

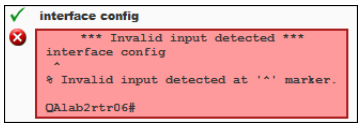
The Job Details Viewer's **Custom Log** page provides the logging information embedded into a script using LOG commands (see the *LOG {-INFO, -WARNING, -ERROR, -DEBUG}* topic for more information).

The Job Details Viewer's **Files** page provides download links for any files associated with the device's script execution. In CCS scripts, files are created and written using the *ARCHIVE* directive. Files can only be associated with each device in the script, and cannot be concatenated into a single large file encompassing the entire script run. Also see *CCS Data Archive and Export* for more information.

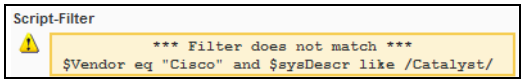
When Errors Occur



When the Job Viewer's Status column shows Errors for the executed Job, it individually lists all devices exhibiting errors. Clicking on an Error link for any device shows the location of the error in the **Process Log**. Errors appear in red. Errors can be due to many issues, including typos in a command directive, incorrect use of an attribute or mistakes in device group assignment.



If a script generates a list of errors, keep the Job Viewer window open, go to **Configuration Management –› Job Management –› Scripts** in NetMRI and edit the script to fix the errors. Back in the Job Viewer, click the **Rerun Errors** button. A form page appears in the window, used to re-run the script against the devices that failed the script execution in the existing job instance. Clicking **Run Now** executes a new Job without the need to go back through the Script Run Now wizard.

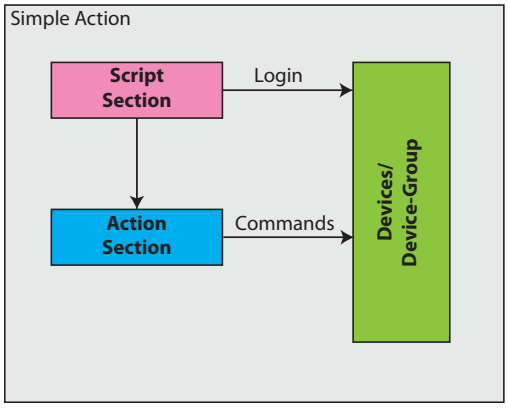


Failure to match a pattern in device output will not generate errors–a **Filter Does Not Match** message appears in yellow in the Process Log page. This typically indicates that a pattern specified in a *Script-Filter:* attribute or *Trigger-Filters:* attribute does not appear in the CLI output generated through the script, or that the pattern is incorrectly specified through a typo or an incorrect regular expression.

Note: Exercise caution when writing and executing CCS scripts. Because error checking is not precise even in the best of cases, it is possible to execute endless loops that may result in needing to reboot the NetMRI appliance to stop the runaway job. Test scripts against a single network device and avoid running scripts against production networks until you are certain the script will have a properly finite execution.

CCS Script Hierarchies

CCS uses four basic building blocks for job automation, which are called *sections* in each CCS script.



- *Script*–the written CCS script, written using a text editor and imported into NetMRI, or written using NetMRI's built-in **Edit Script** feature. The Script section identifies the working set of network devices to be acted on by the CCS script. The script runs only on the devices that the user wants to run it upon. One Script section can be defined per script. The device identification is carried out through subordinate elements such as *variables* and *filters*. The Script section precedes all other section types, including Actions, Issues and Triggers. The simple script workflow example here shows a Script section that enables the script to run on a specific device or device group (by logging in to the device), and then passes the execution on to an Action section, which operates on each logged-in device in turn, sending a batch of CLI commands.
- *Action*–An Action section defines a sequence of CLI commands to be executed by the script on the matching device(s). The CLI commands are derived from the command syntax for the network infrastructure devices involved in the job–Cisco, Extreme, Juniper, and so on. *Any CCS script must contain at least one Action section.*
- *Issue*–An Issue section processes the output of the CLI commands (the responses produced by the device as a response to the executed CLI command) and generates a custom CCS issue. An Issue is similar to a custom type of Trigger (see the following paragraph for more). Those custom issues are added to NetMRI's Issues list (**Network Analysis –› Issues**) and affect the NetMRI appliance's Network Scorecard values.
- *Trigger*–A trigger section responds to CLI output by issuing new sequences of CLI commands to change device configurations, or to generate further output from the network device in an attempt to activate a second trigger. Thus, you can nest triggers in scripts to perform more complex processing.

Other vital elements of the CCS scripting language include the following:

- Variables (see the section *CCS Variables Usage* for more);
- Attributes, which are the parts making up each section (see the topics in the section *Standard CCS Attributes* for more);
- Looping (see the section *Looping with CCS Scripting* for more).

CCS Variables Usage

Scripting and programming languages use *variables* to define important values that allow the NetMRI job engine to respond according to what it sees in the data it's parsing. CCS is no different; in each section of a CCS script (Script, Actions, Issues and Triggers), you use variables to define the values that CCS uses to run CLI commands and to match against output data from network devices.

NetMRI 's job engine recognizes several types of variables:

Well-Known Variables–A set of standard variables universally recognized by the NetMRI job engine in all scripts whereever they are called out. See the section *Well-Known Variables* for a list and brief descriptions of the well-known variables in the NetMRI system.

Script-Variables–User-defined variables that are used in sections of a script to allow interactive user/admin input during script execution. You define `Script-Variables` in the Script section of any script. (See the section *Script-Variables:* for more detail.) When a script runs, a `Script-Variable` prompts the user running the script to enter the value (IP address, user name, device name, etc...).

A `Script-Variable` can refer to lists in the NetMRI system (**Configuration Management –› Job Management** tab **–› Lists**) as an input type. (See and *The getListValue( ) Function* for more detail.)

Trigger-Variables–A special variable type, limited to Trigger sections, that is used to extract pieces of information from the command-line output resulting from an `Action-Command` attribute. Consult the topic *Trigger-Variables:* for more information.

SET variables–Custom, user-defined variables that can be declared only in specific Command scripting attributes. SET variables can also reference lists, which are two-dimensional tables of data that are installed into the Lists page in NetMRI. See *Setting Variables in Command Attributes* for more.)

Note: A longer list of NetMRI-standard scripting variables can be found in the section "Scripting Well-Known Variables (Perl and CCS)" in the *Infoblox NetMRI Administrator Guide* or in online Help.

In CCS scripts, you always declare a variable by using a dollar sign, which is also the sign of a Perl scalar variable:

```
$vendor
```

CCS scripts also reference declared variables with the dollar sign. Setting a value for a variable requires standard arguments:

- `eq` (Equal to)
- `like` (Similar to, used with wildcards)
- `in` (variable value is contained in the following)
- `ne` (not equal to)

Declaring a variable, even if you do not assign it a value, also requires declaring its type, including the possibility of using a list:

- regular expression (example in section *Setting Variables in Command Attributes*)
- `word` (string)
- `boolean`
- `string`
- `list`

Setting Variables in Command Attributes

You can declare a special variable type, called a SET variable, in `Action-Commands` and `Trigger-Commands` attributes of a CCS script. The following examples illustrate how SET variables may be used to keep track of a state.

An `Action-Command` attribute declaration also illustrates the valid use of an equals sign operator (=):

```
Action-Commands:
    SET: $updateMade = "no"
```

In `Action-Commands` and `Trigger-Commands` attributes, a SET variable can refer to a list in NetMRI and reference its data (also see the section *The getListValue() Function*):

```
Action-Commands:{$location eq "West"}
    SET: $log1 = getListValue(NetworkServers,Geo,WS1,NMAddr,null)
```

If the value found in the `$location` variable happens to be West, the script opens the list `NetworkServers` from NetMRI and searches for the specified value.

Using List Variable Types

CCS can reference lists in the NetMRI system (lists are edited and imported in the **Configuration Management –› Job Management** tab **–› Lists** page). The following arguments are used by CCS when declaring list variables in a Script-Variables section:

```
$variableName list "list_name","key_column","key_value","returnvalue_column","default"
```

The value `$variableName` is the name of the variable to which the result of the list lookup will be assigned. The `list` keyword is required. The third argument consists of the following: In the given `list_name`, in the `key_column`, look for the first match of the specified `key_value` from top to bottom of the list. If a match is found, return the value contained in the `returnvalue_column` in the same row in which the match was found, else, return the `default` response. Omitting the optional `default` argument is the same as specifying a default argument of "”; in other words, an empty string.

An example:

```
$oscheck list zoneID,serveraddr,eng.corp100.com,ostype,NOMATCH
```

Here, a list called **zoneID** is present (or presumed to be present) in the NetMRI **Lists** page, instructing to search in the column **serveraddr** for the key_value **eng.corp100.com**, and if a match is found, return the value found in the matching row's **ostype** column into the variable `$oscheck`. If no match occurs, the string NOMATCH is assigned to the variable `$oscheck`.

Also see the section *The getListValue() Function* for more information about using list data in CCS scripts.

Logical Expressions and Regular Expressions

CCS supports logical expressions and regular expressions for more-complex value matching and triggering only when two or more conditions are met.

Consider the following statement:

```
Trigger-Filter:
    $adminMode ne "trunk" and $operMode ne "trunk"
```

The scripting statement calls two variables (`$adminMode` and `$operMode`). In this case, both are *well-known* variables. The logical and operator determines that the match applies only if both variable values (a **show interface** operation listing the administrative mode and operational mode for a switch port) *do not* reflect the stated equivalence of "`trunk`", and because the statement is part of a `trigger-filter`, the corresponding `Trigger-Section` is called by the script.

Valid Boolean logical operators include:

- and (&&)
- or (| |)

The meta characters that can be used to build regular expressions for CCS scripts include the following:

|	OR operator. Provides alternatives for matching. The left side of the | may match, or the right side may. To be meaningful, it must be enclosed by parentheses "( )."
*	Match the preceding item 0 to *n* times, of a character or set of characters. Classic wildcard operator.
[ ]	(Square brackets) Set an inclusive range of values or an inclusive operation within a pattern, or a match against alternatives.
{ }	(Curly brackets) Allows matching against a specific number of occurrences of a pattern, a range defining the minimum and maximum number of instances to match against, and a starting value for n to infinite matches for a pattern. Its function is related to the + and * operators.
( )	Enables matching against one of a set of alternatives, if combined with the OR operator (|).
+	1 to *n* occurrences of a character or set of characters.
^	Boolean NOT operator. Prevents a specified character, delimiter, value or string from being included in a match within the specified pattern.
/<pattern>/	Encloses the pattern/regular expression to be compared against the fetched value in the variable $_.
.	Matches any single character.
" "	Double quote marks enclose the entire pattern to match.

A string variable with a regex may be used to match against multiple instances of similar values in the same variable:

```
Trigger-Variables:
   $ifName "/FastEthernet[^ ]*/"
```

A Perl-style regex is used to define `$ifName`. First, the entire string is enclosed in quote marks ("") to ensure that the entire string, including any spacebar characters (such as you would see in a Cisco `int FastEthernet0/11` command string) is treated as a single parameter when declaring the variable.

```
/FastEthernet[^ ]*/
```

The two forward slashes encapsulate the full string to match against, which contains a regular expression.

```
FastEthernet[^ ]*
```

An extensive discussion of regular expressions is beyond the scope of this document; CCS supports the same conventions for regular expressions as the Perl language.

Standard CCS Attributes

Script Section	Action Section	Issue Section	Trigger Section
Script:	**Action**	**Issue-ID**	**Trigger**
Script-Filter	**Action-Commands**	**Issue-Severity**	**Trigger-Template**
Script-Description	Action-Description	**Issue-Template**	**Trigger-Commands**
Script-Variables	Action-Filter	**Issue-Filter**	**Trigger-Filter**
Script-Timeout	Action-Timeout	**Issue-Details**	Trigger-Variables
Script-Login	Output-Triggers	Issue-Variables	Trigger-Description
		Issue-Description	Output-Triggers

Attributes in bold are mandatory when the section type is used in a script.

Each section type—Script, Action, Issue and Trigger—is written using a series of *attributes* that define how NetMRI processes the section. CCS defines a set of standard attributes used by all scripts to contain defined values, call other script sections, load list data and other functions.

Unlike variables, an attribute may contain multiple values and call other scripting elements. An Action, for instance, can call a Trigger.

Note: All attributes specifications in CCS scripts are case-sensitive–each attribute is cited in initial caps in a script, as in `Action-Filter` or `Issue-Variables`.

Though *attributes* within a section type are mandatory or optional, a specific *section* type is not necessarily required for a script. For example, the **Cisco Set Port Fast** CCS script (found in **Configuration Management –› Job Management** tab **–› Scripts**) contains two Action sections and two Trigger sections, with no Issue section.

Standard CCS attributes are described in the following subsections.

Script Section Attributes

Every CCS script must have a Script section, which defines the devices on which the script will execute, and can contain other script features such as new variables for use by other parts of the script; a timeout period; and any possible admin login information needed by the devices for which the script runs against. Script sections are defined using the following attributes:

Attribute	Status	Purpose	UI Field
`Script:`	(required)	defines the script name*‡	**Name**
`Script-Filter:`	(required)	defines the type of devices to be processed	**Script Run Now Wizard**
`Script-Description:`	(optional)	describes the script purpose*‡	**Description**
`Script-Timeout:`	(optional)	define a time period for the script to wait for output	N/A
`Script-Variables:`	(optional)	input values when executed through the Run Now feature	N/A
`Script-Login:`	(optional)	True or False, default is True if not used	N/A

*These attributes are specified in fields in the Edit Script dialog. They are present as separate lines when the script is viewed in text form after exporting. If you create a script in a text editor and import it into NetMRI, these lines are transcribed into the corresponding fields in the Edit Script dialog.

‡These attributes are described here for reference. If entered in a script, an error is generated when the script is saved. Use the UI to define these values. For reference, define these attributes and comment them out in the script.

The `Script` and `Script-Filter` attributes are mandatory.

Script:

Used In: Script sections

Status: Mandatory

The `Script` attribute is normally entered in the **Name** field in the **Edit Script** dialog. Information here is also provided for cases when a script is created in an external text editor, then imported. The Script attribute can contain any text characters.

Example

```
#Script: Example 4 –> Cisco Set Duplex
```

Script-Description:

Used In: Script sections

Status: Optional

This attribute is normally entered in the **Description** field in the **Edit Script** dialog. Information here is useful when a script is created in an external text editor, then imported into NetMRI.

You use the `Script-Description` attribute to provide a description of the script. The attribute can contain any number of plain text lines (no HTML tags).

```
#Script-Description:
#This script sets the duplex to auto on all Fast Ethernet
#interfaces on all Cisco switches.
```

Script-Filter:

Used In: Script sections

Status: Mandatory

A script attribute that defines the devices that will be acted upon by the CCS script. When a script runs, the `Script-Filter` ensures that no matter what device group is chosen, the script will run only on the devices that the user wants.

Use the `Script-Filter` attribute to specify the set of network devices (by IP address range, model type, NetMRI Device Group, etc...) to be acted upon by the CCS script during execution. Filtering uses the same syntax used for defining device groups (see the *Understanding Group Membership Criteria* topic in online Help for details). Filtering uses variables to specify the data types and value ranges required by the script. To tie more than one variable into a `Script-Filter` attribute, use Boolean operators such as “&&” (“and”), and the “|” character (“or”).

You can also nest logical operators using parentheses ( ). For some variables, you can use multiple matches enclosed by square brackets “[ ]”.

Example

A single AND operator is used.

```
Script-Filter:
    $Vendor eq "Cisco" and $Model in ["2811", "2821", "871", "2621XM"]
```

A filter can use multiple AND operators, or whatever logic is necessary for filtering.

```
Script-Filter:
    $Vendor eq "Cisco"   and
    $sysDescr like /IOS/ and
    $Version like /^1[2-9]/ and
    $Type in ['Router', 'Switch-Router']]
```

A more complex Boolean operation is used, with parentheses encapsulating arguments.

```
Script-Filter:
($Vendor eq "Cisco" && $Model in ["2811", "2821", "871", "2621XM"]) or
($Vendor eq "Extreme" and $sysDescr like /XOS/)
```

Script-Login:

Used In: Script sections

Status: Optional

The `Script-Login` attribute is available in the Script header section for both CCS and Perl scripts. This attribute specifies whether the job engine should automatically establish a connection with the target device. The only valid values for this attribute are “true” or “false”. Should the `Script-Login` attribute not be specified, the value defaults to “true”. You will note that most CCS scripts in NetMRI do not use this attribute, ensuring that NetMRI uses the credentials in its database to log in to the target devices.

For CCS scripts, the value is respected. That is, if the value is “true” a connection will automatically be established with the target device. If the value is “false” no connection is established with the target device.

Example

```
Script-Login:
    false
```

If you want to hide the password string and replace it by stars characters ****** in logs, use the <hidden></hidden> tag:

```
Action-Commands:
    config terminal
    username $username password 0 <hidden>$password</hidden>
    exit
    write memory
```

Script-Timeout:

Used In: Script sections

Status: Optional

You use the `Script-Timeout` attribute to define the desired period of time for which the script will wait for completion of any directive to a device. When a command finishes executing on a network device, it returns the prompt back to the session. CCS watches for this before attempting further interaction with the device in question. By default, `Script-Timeout` values are set to 60 seconds in the NetMRI system; if the executed device command does not return the prompt back to CCS within 60 seconds, you will receive a timeout error from CCS.

The maximum timeout value is 300 seconds. Setting this value will globally apply to the entire CCS script unless you change an *Action-Timeout:* value, which overrides the global `Script-Timeout` value for that Action section.

Also see *Action-Timeout:* for a corresponding timeout value that can be set for individual sections of a script.

Example

```
Script-Timeout: 300
```

Script-Variables:

Used In: Script sections

Status: Optional

Note: `Script-Variables` are not allowed if the `Script-Schedule` attribute is defined in the same section, because scheduled scripts must have all necessary data defined for them at runtime.

You use the optional `Script-Variables` attribute to define data input prompts to be displayed to the user when manually executing a CCS script. This is how you allow user input into an executed CCS script. The CCS script uses the values entered by the user as it executes. The Script-Variables attribute requires one or more variable definitions. Each variable must be defined on its own line, as in the following example:

```
Script-Variables:
$usernameword "User Name"
$password "New Password" string
```

The exact format of a `Script-Variable` definition is:

`<variableName> <inputType> <defaultValue> <evalType>`

`<variableName>` can be any string (without blanks) that starts with $.

Note: Variable names are case-insensitive. Script variables can be referenced from anywhere in the script, but only in lower case.

`<inputType>` determines the type of HTML entry used in the script input form and enforces input validation on values entered into the form. For example, to ensure that an IP address in dotted notation (or hostname) is entered, declare a variable with `<inputType>` of `<ipaddress>`. The following inputTypes are supported, with the associated HTML entry fields:

inputType	Corresponding HTML	Input method notes
`string, word, int, integer, double, datetime, duration, url, id, phoneno, zipcode, email, ipaddress, regular ssi`	Text	One line of text can be entered in the input form field
`list`	listName (from NetMRI), ColumnLookupName, KeyValueName, ReturnColumnValue, DefaultName	Use a separately defined list (see *The getListValue( ) Function* section for more information).
`password`	Password	Same as Text, except all characters entered into the form are obscured
`text`	Textarea	Multiple lines of text can be entered in the input form field
`boolean`	check box	If checked, value sent to server is “on”; Otherwise it is “off”

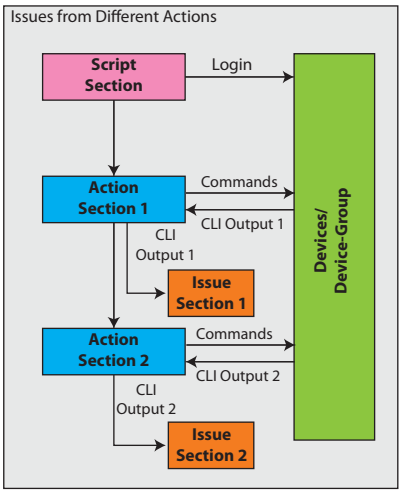
When first shown, `<defaultValue>` appears as the variable value in the script input form. You can overwrite the value as needed before submitting the form. `<EvalType>` specifies how the value should be evaluated when used in a logical filter expression. The `<evalType>` can be specified as string (this is the default if nothing is specified), number, or ipaddr. For example, to look for interfaces where `$ifIndex < 10`, specify “number” as the `<evalType>` for the `$ifIndex` variable.

Note: By default, all variables are evaluated as strings in filter expressions.

Example

```
Script-Variables:
    $change_location     boolean
    $new_location        string      "Enter location here"
    $add_community       boolean
    $new_community       string      "Enter community here"
```

Action Section Attributes



Action sections send CLI command strings to specified devices. Scripts use the Action section to execute CLI commands to set properties on a network device, or to generate output to be used by the Trigger or Issue sections. At least one Action section must be defined within any CCS script, and more than one Action section may be defined in a script. Action sections also may be used in combination with other attributes. The figure to the left indicates that multiple Action sections may be defined in a script.

Based on the response from each device across the CLI session, it’s possible to make decisions on that CLI output, based on filtering that information to see if it matches a particular value or result; calling another Action as a result; calling an `Output-Trigger` section of the script to execute a more-complex operation such as a loop (for example, to check a large number of interfaces on a single switch; see *Trigger Section Attributes* for more information), or if a particular state is shown in the CLI output as the result of the Action, display an Issue in NetMRI. (The section *Issue Section Attributes* shows an example.)

Attributes supported by the Action section include:

Attribute	Status	Purpose
`Action:`	(required)	name of Action
`Action-Description:`	(optional)	describes the Action
`Action-Filter:`	(optional)	defines when the Action should be executed
`Action-Commands:`	(required)	defines the device commands to be executed
`Action-Timeout`	(optional)	define a time period for the script to wait for output
`Output-Triggers:`	(optional)	defines how to process the output stream
`Action-Prompt:`	(optional)	defines the new prompt on an action

Action:

Used In: Action sections

Status: Mandatory

`Action` is a required attribute used to name the Action. This name must be unique within the CCS script to allow the definition of other Action attributes.

Example

```
Action: Set User Password
```

Action-Description:

Used In: Action sections

Status: Optional

The `Action-Description` attribute provides descriptive text for the current action section.

Example

```
Action-Description:
This script sets the password for certain pre-determined
user accounts to the same password.
```

Action-Commands:

Used In: Action sections

Status: Mandatory

The `Action-Commands` attribute is used to define a series of one or more command strings, entered one per line, to execute for each device that meets the criteria established in the `Action-Filter`.

Example

```
Action-Commands: show interfaces
```

You can extend `Action-Commands` to specify optional filter criteria, enclosed by curly brackets { } that restrict execution to cases where specific conditions exist. Example:

```
Action-Commands: { $Vendor eq "Cisco" }
   show interfaces
```

Besides CLI commands, the `Action-Commands` attribute accepts the `sleep` directive, which pauses script execution for a specified number of seconds. Syntax for this directive is:

```
sleep: <numberOfSeconds>
```

This example shows how a script can be paused for 10 seconds:

```
Action-Commands:
... <some commands>
sleep: 10
... <more commands>
```

SET variables may be defined in `Action-Commands`:

```
Action-Commands:
    SET: $runscript = "no"
    SET: $nofireissue = "no"
    SET: $do_acl = ""
    SET: $do_type = ""
    SET: $do_clear = "no"
```

Action-Filter:

Used In: Action sections

Status: Optional

The `Action-Filter` attribute is similar to the `Script-Filter` discussed in *Script Section Attributes*, with the exception that an `Action-Filter` can reference SET variables. The purpose of the `Action-Filter` attribute is to include or exclude certain devices from execution by the `Action-Commands` attribute defined in the Action section.

If the filter evaluates to "true," the script executes subsequent `Action-Commands`. If the filter evaluates to "false", further `Action-Commands` will not execute. An absence of this attribute within an action section means the filter will automatically be set to "true" and the `Action-Commands` execute for all devices.

Any SET variable defined in this attribute can be referenced by following parts of the CCS script.

Example

```
Action-Filter:
    $runscript ne "no" and $do_acl ne "" and $do_contents ne "" and
    $change_ok eq "yes"
```

Action-Timeout:

Used In: Action sections

Status: Optional

You use the `Action-Timeout` attribute to enforce a specific timeout value for a particular Action section in a script. By default, `Action-Timeout` inherits its value from the value set for `Script-Timeout`. When a command finishes executing on a network device, it returns the prompt back to the session. CCS watches for this before attempting further interaction with the device in question. If the executed device command does not return the prompt back to CCS within 60 seconds, you will receive a timeout error from CCS. `Action-Timeout` values inherit their default settings from the global `Script-Timeout` value in the NetMRI system. The maximum timeout value is 300 seconds. Setting this value in a specific Action section will override the global `Script-Timeout` value for that Action section.

Example

```
Action-Timeout: 300
```

Output-Triggers:

Used In: Action sections

Status: Optional

Use the optional `Output-Triggers` attribute to specify additional processing after the `Action-Commands` attribute has been executed. `Output-Triggers` can be used to continue processing with one or more additional Triggers or Issues.

`Output-Triggers` can perform many independent actions on the output of `Action-Commands`. Below, two additional Triggers are called in order; when finished, the script calls an Issue.

Example

```
Output-Triggers:
    Process Down Interfaces <- Trigger
    Process Up Interfaces <- Trigger
    Generate Initial Interface Issue <- Issue
```

Action-Prompt:

Used In: Action sections

Status: Optional

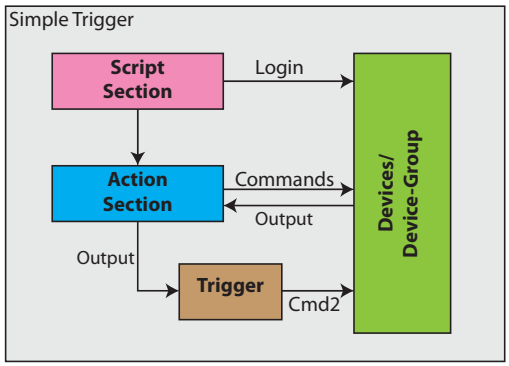
Use the `Action-Prompt` attribute to override the device prompt when the command(s) executed in the `Action-Commands` attribute change the prompt on the device.

Note: The default CCS scripting engine works for many of the cases.

Example

```
Action:        show run 1
Action-Commands:
    sh run
    Action:         Create ACL
#here we need the 3 prompt to handle the 3 cases including exit
Action-Prompt: /(DEVrtr01\(config-ext-nacl\)#)|(DEVrtr01\(config\)#)|(DEVrtr01#)/



Action-Commands:
     conf t
    ip access-list extended test2
    exit
```

Trigger Section Attributes



Trigger sections are optional but can add a lot of flexibility to a CCS script. They are used to process output from CLI commands initially executed by another Action (or possibly another Trigger) section. (The figure to the left shows an Action calling a Trigger section.) Triggers can be used to process the CLI output from the first batch of CLI commands in an Action section, by executing another batch of more-detailed commands on the same device currently being processed. Such detailed commands may be used to extract additional information required for later Actions, Triggers or Issues; among the possibilities is to change the device configuration to actually fix a problem.

Triggers perform "IF –› THEN" logical comparisons of the results of CLI commands, iterating over all occurrences of a given pattern in the output stream. Triggers are the only looping mechanism provided in CCS. They can call other Triggers, allowing multiple levels of logical checks before taking final action.

The primary difference between an Issue section and a Trigger section is that the Trigger executes a set of commands compared to an Issue which generates an Issue for NetMRI to report. They both use variables and templates to define patterns of interest, and they can both use filter expressions to determine if something needs to be done.

Triggers support the following attributes:

Attribute	Status	Purpose
`Trigger:`	(required)	Trigger name
`Trigger-Template:`	(required*)	identify patterns in output stream for processing
`Trigger-Filter:`	(required*)	defines when commands should be executed
`Trigger-Commands:`	(required)	defines the device commands to be executed
`Trigger-Description:`	(optional)	displayed as help text in the Process Log
`Trigger-Variables:`	(optional)	define/store values extracted from output stream
`Trigger-Prompt:`	(optional)	defines the new prompt on a trigger
`Output-Triggers:`	(optional)	defines how to process the output stream

*Trigger-Template *and/or* Trigger-Filter must exist in a Trigger section.

Trigger:

Used In: Trigger sections

Status: Mandatory

Defines the trigger type. The preceding Action sections call the value defined in the `Trigger:` attribute, using the `Output-Trigger` attribute. All trigger names must be unique in the current script.

Example

```
...
Output-Triggers:
   Find ACL
##############################################################################
Trigger:
   Find ACL
```

Trigger-Description:

Used In: Trigger sections

Status: Optional

Optional attribute to describe the functions for the current trigger.

When CSS detects the `Trigger-Description` section, the following description is not treated as code and is otherwise ignored.

Example

```
Trigger-Description:
   The following Trigger-Template is applied to the output stream
   of the previous Action-Commands to determine if the following
   Trigger-Commands should be executed.
```

Trigger-Template:

A text snippet that defines the command syntax to match against the output from the device. You can call values from variables in a `Trigger-Template` to flexibly match multiple instances or to detect a particular value.

When NetMRI's CCS scripting engine receives the output from the executed command sequence, CCS references the first output trigger to process the command output. Within the trigger, a segment of command output, using variables and wildcards if necessary, ranges over the output stream and executes an associated set of **Trigger-Commands** if necessary (see following section).

Example

```
Trigger-Template:
    username $username password 0 .*
```

This example shows Cisco but can be for any vendor supported by NetMRI.

The `Trigger-Template` provides a pattern matching window that is applied across the most recent CLI command output stream. The template matches characters in the command output zero to possibly many times. If no matches occur, the script stops the execution of the trigger. If one or more template matches is found, the trigger continues processing the `Trigger-Filter` for each match.

Example

Consider the following `Trigger-Template` which uses declared `Trigger-Variables` (`$ifName and $ifDuplex`); note the use of the "+" operator, which matches against 1 to *n* instances of a set of characters:

```
Trigger-Template:
    [[$ifName]] is.+, line protocol is.+
    ...
    Keepalive.+
    [[$ifDuplex]]-duplex.+
```

and the following output stream:

```
0 lost carrier, 0 no carrier
0 output buffer failures, 0 output buffers swapped out
FastEthernet0/1 is up, line protocol is up
Hardware is Fast Ethernet, address is 0002.b9fc.b701
MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,
reliability 255/255, txload 1/255, rxload 1/255
Encapsulation ARPA, loopback not set
Keepalive not set
Auto-duplex (Half), Auto Speed (10), 100BaseTX/FX
ARP type: ARPA, ARP Timeout 04:00:00
Last input 00:00:00, output 00:00:00, output hang never
Last clearing of "show interface" counters never
```

The `Trigger-Template` matches the previous command output stream, because the Template contains several regular expressions and fixed patterns.

Because the `Trigger-Template` matches at least once, the processing of the Trigger continues. Because the Template uses previously defined `Trigger-Variables` (within double-brackets "[[ ]]"), the script extracts corresponding values from the command output stream for use in the `Trigger-Filter`. For example, `$ifName` now contains the value **FastEthernet0/1** and `$ifDuplex` contains the value **Auto**.

The pattern matching notation ". . ." appearing in the above `Trigger-Template` means "match 0 or more lines" and is used to skip over multiple lines of text while scanning for the next occurrence of a pattern (e.g., the text **Keepalive**).

Trigger-Commands:

Used In: Trigger sections

Status: Mandatory

The Trigger-Commands attribute enables substantial follow-up actions to be taken by a Trigger section following an Output-Trigger call. The attribute allows the script author to execute a sequence of CLI commands for all template matches that pass the Trigger-Filter criteria. Each command must appear on a new line. You can use Conditional processing with { } notation (similar to Action-Commands in the *Action-Commands:* topic) in Trigger-Commands as well.

Besides CLI commands, the Trigger-Commands attribute accepts sleep directives, which pause script execution for a specified number of seconds.

```
sleep: <numberOfSeconds>
```

Example

```
Trigger-Commands:
    config terminal
        interface $ifName
            spanning-tree portfast
            exit
        exit
    SET: $updateMade = "yes"
```

Example

```
Trigger-Commands: { $priority eq "0" and $state ne "Ready" }
    SET: $matchFound = "yes"
...
```

In both cases, the SET variables are newly declared in the script. It forms a basic IF-THEN loop, in this case where IF the device's priority value is "0" and the device's state is determined to not be READY, the new matchFound variable is set to "yes" and further script directives designed to respond to the state will execute.

That SET variable also can be referenced anywhere else as needed in the script after declaration.

Note: This example is in a Trigger section. A Trigger-Command is the only attribute type that can execute any type of looping or iterative processing in a CCS script. For more, see the *Trigger-Commands:* section of this Guide.

Trigger-Filters:

Used In: Trigger sections

Status: Optional

The Trigger-Filters attribute is used to determine if Trigger-Commands and Output-Triggers attributes in the Triggers section of the script should run. If no Trigger-Filter exists, the script runs for every match identified by the Trigger-Template.

Trigger-Filters can contain Script, Trigger and SET variables and apply to every template match instance.

Example

```
Trigger-Filter:
    $adminMode ne "trunk" and $operMode ne "trunk"
```

Example

In this case, a match is made with a boolean OR:

```
Trigger-Filter:
    $ifName eq "FastEthernet0/0" |
    $ifName eq "FastEthernet0/1"
```

Trigger-Prompt:

Used In: Trigger sections

Status: Optional

Use the `Trigger-Prompt` attribute to override the device prompt when the command(s) executed in the `Trigger-Commands` attribute change the prompt on the device.

Note: The default CCS scripting engine works for many of the cases.

Example

```
Trigger: acl test3
Trigger-Template:
    ip access-list extended test2
Trigger-Prompt: /(DEVrtr01\(config-ext-nacl\)#)|(DEVrtr01\(config\)#)|(DEVrtr01#)/
Trigger-Commands:
     conf t
     ip access-list extended test3
     exit
```

Trigger-Variables:

Used In: Trigger sections

Status: Optional

CCS scripts use the `Trigger-Variables` attribute to define "patterns of interest" found within the output generated by CLI commands. `Trigger-Variables` are used in `Trigger-Templates` to extract certain values from command output to be used in determining if certain conditions exist on the device. The format of `Trigger-Variables` is as follows:

```
<variableName> <extractPattern> <evalType>
```

`<VariableName>` can be any name (without blanks) that starts with $. Variable names are case-insensitive and can be accessed anywhere in the script.

`<ExtractPattern>` can be specified using any of the keywords listed below, or specified as a regular expression.

extractPattern	Corresponding Regular Expression
String	/[^\r\n]+/
Word or Id	/\w+/
Int or Integer	/\d+/
Double or Float	/\d+\.\d+/
Datetime	/\d{2,4}-\d{1,2}-\d{1,2} \d{1,2}:\d{1,2}:\d{1,2}/
Phoneno	/\(?\d{3}\)?[-]?\d{3}-\d{4}/
Zipcode	/\d{5}(-\d{4})/
Email	/[\w\.-]+@[\w\.-]+/
Ipaddress	/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/
Regular expression	/Any valid Perl 5 regular expression/

`<evalType>` specifies how the variable value should be evaluated when used in a logical filter expression, as a string (this is the default if nothing is specified), a number, or an ipaddr.

For example, to look for interfaces where $ifIndex < 10, specify `int` as the <evalType> for the `$ifIndex` variable. By default, all variables are evaluated as strings in filter expressions.

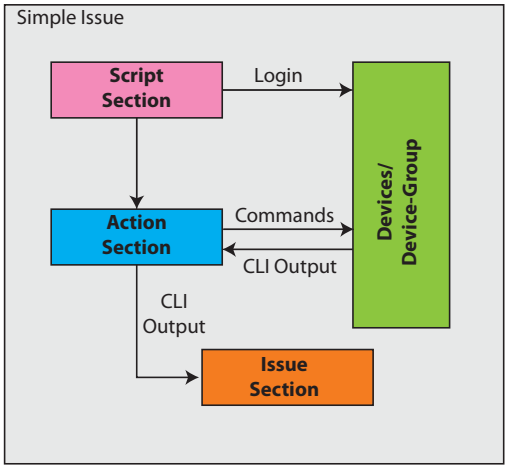
Example

```
Trigger-Variables:
   $ifName   String
   $ifIndex /\(?\d{3}\)?[ -]?\d{3}-\d{4}/ string
   $phoneno  Word
```

Example

```
Trigger-Variables:
   $ifName /FastEthernet.*/
Trigger-Template:
   interface [[$ifName]]
```

Issue Section Attributes



Use one or more optional Issue Sections to generate custom NetMRI issues based on criteria encountered in the processing of the script. NetMRI uses Issues to quantify and report problems and events across the network. In the **Network Analysis –› Issues** tab, the **Network Scorecard** shows the results of the daily analysis process and all issues generated for the latest time period. Issues can also be filtered. More than one Issue may be referenced within a CCS script. They can use variable and template definitions to define patterns of interest, and they can both use filter expressions to further examine output to decide when something must be done (the Issue is reported). Three types of Issues are reported by NetMRI, along with a special fourth CCS Issue classification.

- **Errors** are important issues that may affect the smooth operation of the network. Generally, such issues are clear signs that something is wrong.
- **Warnings** are intermediate level issues that should be addressed after the errors have been corrected. A warning may not be a real problem, depending on the design and operation of the network.
- **Info** issues are provided for information, and typically alert you to minor events that may or may not indicate a problem.
- **CCS** issues are directly related to the execution of a CCS script. CCS uses the Issues page for notification after a script executes. An example is **Invalid User Account**, which reports the mistaken use of a non-Admin user account or an incorrect login for a device.

Issue sections support the following attributes:

Attribute	Status	Purpose
`Issue:`	(required)	issue name
`Issue-ID:`	(required)	system-wide unique issue ID
`Issue-Severity:`	(required)	“Error”, “Warning” or “Info”
`Issue-Template:`	(required*)	identifies patterns in the output stream
`Issue-Filter:`	(required*)	defines when the issue should be generated
`Issue-Details:`	(required)	detailed issue data

`Issue-Description:`	(optional)	issue description
`Issue-Variables:`	(optional)	define/store values extracted from output stream

*Issue-Template *and/or* Issue-Filter must exist in a Trigger section.

The list of attributes in the Issues section declares the issue that will be generated should a positive match occur, which will appear in the **Network Analysis –› Issues** page and in configured notifications.

Issue:

Used In: Issue sections

Status: Mandatory

Defines the custom issue (previously defined in the NetMRI set) to be generated should a positive match result. This title appears as the banner for the Issue when it appears in NetMRI.

Example

All text after the `Issue:` statement defines the Issue type:

```
Issue: Process All User Accounts
```

Issue-Description:

Used In: Script sections

Status: Optional

Defines the description for a custom issue.

Example

```
Issue-Description: The new firewall rule fails to be written when attempting provisioning
by SDC. Check your device configuration and admin permissions.
```

Issue-Details:

Used In: Issue sections

Status: Mandatory

The mandatory `Issue-Details` attribute defines the name/value pairs to be included in the description of the issue that gets generated. These names are established when the issue template is originally created in the **Configuration Management –› Job Management** side tab **–› Custom Issues** page.

Example

```
Issue-Details:
   Host   $IPAddress
   Name   $Name
```

Issue-Filter:

Used In: Issue sections

Status: Mandatory

CCS requires the `Issue-Filter` attribute if the `Issue-Template` attribute is not provided in the script. The `Issue-Filter` applies additional criteria to determine if the issue should be generated. It uses the same logical operators as for `Script-Filters` and other attributes.

Example

Note the use of Boolean operators in the filter attribute:

```
Issue-Filter:
   $nofireissue ne "yes" and
   ($do_domain eq "yes" and $have_new_domain ne "yes") or
   ($do_nameserver eq "yes" and $have_new_nameserver ne "yes") or
   ($do_netbios_ns eq "yes" and $have_new_netbios_ns ne "yes") or
   ($do_tacacs eq "yes" and $have_new_tacacs ne "yes") or
   ($do_tftp eq "yes" and $have_new_tftp ne "yes") or
   ($do_time eq "yes" and $have_new_time ne "yes")
```

Issue-ID:

Used In: Script sections

Status: Mandatory

The Issue-ID value must be defined in NetMRI's Custom Issue list (**Config Management –› Job Management** tab **–› Custom Issues**) before it can be called out by a CCS script.

Example

```
Issue-ID: InvalidUserAccount
```

Issue-Severity:

Used In: Issue sections

Status: Mandatory

The `Issue-Severity` attribute defines the importance of the issue in the script. Valid choices are Error, Warning and Info. `Issue-Severity` influences the generated Issue's position in the NetMRI Issues List due to the penalty weights assigned by the appliance. For example, a severity of Error yields a placement with other critical Errors closer to the top of the Issues List than Warnings and Informational Issues.

Example

```
Issue-Severity: Error
```

Issue-Template:

Used In: Issue sections

Status: Mandatory

A text snippet that defines the syntax to match against the output for the issue reporting. When NetMRI's CCS scripting engine receives the output from the executed command sequence, it interprets the data against the template, which acts as a regular expression to determine whether the pattern matches or does not.

Example

```
Issue-Template:
   username $username password 0 .*
```

This example shows Cisco but can be for any vendor supported by NetMRI.

Issue-Variables:

Used In: Issue sections

Status: Optional

You can define variables for matching in the `Issue-Variables` attribute. Doing so helps build the patterns of interest that the script searches for in the output. CCS interprets the `Issue-Variables` and `Issue-Template` attributes as regular expressions that are applied to the output stream. Many CCS scripts will not define `Issue-Variables` attributes as frequently as other attribute types. The below example declares a Boolean statement that if a set of three well-known variables are set to a certain state (`$nofireissue` does not equal "Yes," and the variables `$do_domain eq "yes"` equals "Yes" and `$have_new_domain` does not equal "Yes," then an issue will be reported.

Example

```
Issue-Filter:
   $nofireissue ne "yes" and
   ($do_domain eq "yes" and $have_new_domain ne "yes")
```

CCS Data Archive and Export

CCS Data Archive/Export enables archive and export functions for any information available via the CLI. You can send any command output to an external file, which is automatically readable through NetMRI Job Viewer from a file name that you specify.

Data archiving takes place using the ARCHIVE keyword. Archived data can be exported from NetMRI using HTTP.

ARCHIVE

CCS attributes `Action-Commands` and `Trigger-Commands` support the ARCHIVE keyword, used to save the output of various CLI commands into one or more files. In its simplest form:

```
Action-Commands:
   ARCHIVE: sh ver
```

The ARCHIVE keyword creates a file named *device_id-1.log* for each device. The files can be viewed in the **Files** tab of NetMRI's Job Viewer.

The keyword can be followed by an optional file name and should precede all commands for the archived output:

```
Action-Commands:
   ARCHIVE (config.txt): show running-config
```

stores the output of the **show running-config** command in the file *config.txt*.

If a job runs against two or more devices and the script specifies a static file name, all iterations will use the same file name and only one set of data will be present in the Zip archive, containing the last device's output, which will have overwritten all previous output. At the end of the job, the Zip file contains a file with a single set of data from one device (e.g. the last device that the script was run against). This is not particularly helpful.

Use variables to dynamically specify the name of the file:

```
Action-Commands:
   ARCHIVE ($ipaddress.txt): show interface summ
```

stores the output of the **show interface summ** command in a series of files representing each of the devices against which the script runs after the `script-filter` is applied.

Output from multiple commands can be stored in the same file:

```
Action-Commands:
   ARCHIVE ($ipaddress.txt): show interface summ
   ARCHIVE ($ipaddress.txt): show running-config
```

Two successive ARCHIVE directives in the same attribute store the output of the commands **show interface summ** and **show running-config** in the series of files named *‹$ipaddress›.txt*.

At the end of a job, files created using the ARCHIVE keyword are placed in a Zip file. You can view the files and download the Zip file in the **Files** tab of the **Job Viewer**. If you use a variable to define the file names generated by the script, each device in the Job has its own separate output file which is present in the Zip file.

CCS Data Export

The most recent archive Zip file is found in the **Files** tab in the **Job Viewer**, and is placed in a global location where it can be accessed via an HTTP transaction. Details are as follows:

Request Info
URL: //netmri/ccs/tx/common/GetArchive.tdf
Parameters: N/A

Response Info
Content-Type: application/zip
Content-Disposition: NetMRI_CCS_Archive.zip
Content: The most recent ARCHIVE.zip file

Use this mechanism to export the most recent archive.zip file to an external application, or a server, on a scheduled basis using a tool such as wget.

CCS Scripting Commands

Along with CCS attributes and sections, CCS scripting provides a limited number of discrete command functions.

DEBUG

Provides a simple tool to determine whether a statement in a script will run before you actually execute the script. The Process log displays `DEBUG` can be used in `Action-Command` attributes and `Trigger-Command` attributes.

Example

```
Trigger-Commands:
    config terminal
    interface $ifName
    DEBUG: duplex auto
    exit
    exit
```

When the script statement containing the DEBUG executes, a special debug icon appears next to the statement. This indicates that the statement would have run if it had not been removed from execution with the DEBUG: keyword. This tool is useful for cases where you are generating multiple iterations of a particular command and need to see the execution pattern.

Note: Because CCS does not have an integrated development environment, we recommend using DEBUG statements to check for endless loops and logic errors before executing scripts against a live network.

GET-CONFIGS

The GET-CONFIGS directive An optional [MODE] can be specified to the right of the ":" to indicate whether synchronous or asynchronous behavior is desired. Valid values for [MODE] are "synchronous" and "asynchronous". If not specified, [MODE] defaults to "synchronous". Synchronous behavior implies that the script will block until the GC operation has completed (i.e. the operation terminates with an OK status) or abort if an error is encountered (i.e. the operation terminates with an Error status). Asynchronous behavior implies that the script will continue processing after the GC request is initiated. An example follows:

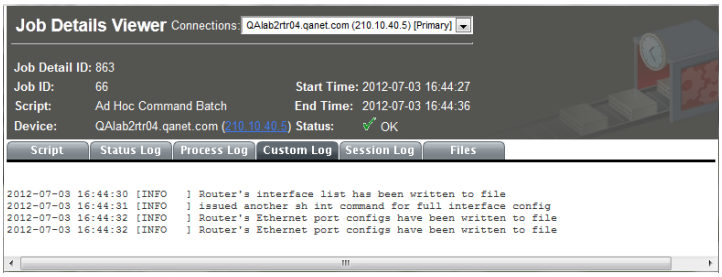
Example

```
# Ensure that NetMRI has the most up-to-date configurations
# on file. The script will block until the Get Configs operation has
# completed since the default synchronous mode is used.


     # Ensure that NetMRI has the most up-to-date configurations on file. The script
     # will block until the Get Configs operation has completed since the
     # synchronous mode is used.

     GET-CONFIGS:

     # Modify the interface description for Fa0/1.

     config t
     interface Fa0/1
              description Get Configs Test
     end

     # Request another Get Configs operation to audit the above change. Since this
     # is the end of the script and there is no need to block until the Get Configs
     # operation has completed, the asynchronous mode is used.


       GET-CONFIGS: asynchronous
```

After script execution completes, output similar to the following can be seen in the Status Log of the Job Details Viewer:

```
+++ 1. Action: Get Configs Test
+++ 1. [Action-Commands]
++ Requesting on demand configuration collection (synchronous) ........ OK
+++ Received TrackingID 1 ............................................. OK
+++ Getting the status of TrackingID 1 ................................ PENDING
+++ Sending 'Keep Alive CR/LF' ....................................... OK
+++ Getting the status of TrackingID 1 ................................ QUEUED
+++ Sending 'Keep Alive CR/LF' ....................................... OK
+++ Getting the status of TrackingID 1 ................................ QUEUED
+++ Sending 'Keep Alive CR/LF' ....................................... OK
+++ Getting the status of TrackingID 1 ................................ RUNNING
+++ Sending 'Keep Alive CR/LF' ....................................... OK
+++ Getting the status of TrackingID 1 ................................ OK
+++ 1. Sending 'config t' ............................................. OK
+++ 1. Sending 'interface Fa0/1' ..................................... OK
+++ 1. Sending 'description Get Configs Test' ........................ OK
+++ 1. Sending 'end' ................................................. OK
+++ Requesting on demand configuration collection (asynchronous) ....... OK
+++ Closing session ................................................. OK
*** Successfully ran configuration command script ***
```

LOG {-INFO, -WARNING, -ERROR, -DEBUG}



The `LOG` directive logs a message of the severity specified by the script. Four levels of severity can be stated, in escalating order: INFO, WARNING, ERROR and DEBUG.

`LOG` can be used in `Action-Command` attributes and `Trigger-Command` attributes. If you try to use `LOG` in other attributes, it will generate a script Error and appear in red in the Job Details Viewer's **Process Log** page.

Log messages are written to the **Custom Log** tab in the Job Details Viewer window.

Example

```
Action-Commands: { $Vendor eq "Cisco" and $type eq "Router" }
   ARCHIVE ($ipaddress.txt): show int summ
   LOG-INFO: Router's interface list has been written to file
```

EXPR

Command-line utility to evaluate regular expressions. `EXPR` can be used in `Action-Command` attributes and `Trigger-Command` attributes. `EXPR` uses a set of arithmetical and Boolean operators to perform simple evaluations between two arguments. `EXPR` is also used to set the integer value.

`ARG1 | ARG2`	ARG1 if it is neither null nor 0, otherwise ARG2
`ARG1 & ARG2`	ARG1 if neither argument is null or 0, otherwise 0
`ARG1 < ARG2`	ARG1 is less than ARG2
`ARG1 <= ARG2`	ARG1 is less than or equal to ARG2
`ARG1 = ARG2`	ARG1 is equal to ARG2
`ARG1 != ARG2`	ARG1 is unequal to ARG2
`ARG1 >= ARG2`	ARG1 is greater than or equal to ARG2
`ARG1 > ARG2`	ARG1 is greater than ARG2
`ARG1 + ARG2`	arithmetic sum of ARG1 and ARG2
`ARG1 - ARG2`	arithmetic difference of ARG1 and ARG2
`ARG1 * ARG2`	arithmetic product of ARG1 and ARG2
`ARG1 / ARG2`	arithmetic quotient of ARG1 divided by ARG2
`ARG1 % ARG2`	arithmetic remainder of ARG1 divided by ARG2

Example

```
# Add 1 to the variable $counter
expr: $counter = $counter + 1
# Compute the remainder of the variable $counter divided by 2
expr: $remainder = $counter % 2
# Assign the integer value to the variable $name
expr: $name = 6
```

The getListValue( ) Function

A function for a CCS script to look up values in a list. The list must be installed in the NetMRI appliance (**Configuration Management –› Job Management** tab **–› Lists** page) to be referenced by scripts. For the given list name and the first specified column in the lookup, CCS finds the first row containing a defined matching value, and fetches the corresponding row's value from another data column in the same data record.

The parameters in `getListValue( )` are the following:

```
getListValue(<list_name> <key_column> <key_value> <value_lookup_column> <default>)
```

`<list_name>`	The name of the list in the NetMRI Lists page.
`<key_column>`	The list column to search for the matching value in `<key_value>`. The specified column can be any column in the table.
`<key_value>`	The value to search for in the first `<key_column>` list column.
`<value_lookup_column>`	The location in the matching data record, in the list, to look up the value.
`<default>`	The default message should no matching value appear in the list for `<key_value>`. If specifying no `<default>`, the argument `null` must be used. An example of a default message would be NOT_FOUND.

Example

```
Action-Commands:{$location eq "West"}
   SET: $syslog1 = getListValue(NetworkServers,Server,WS1,IPAddr,null)
   SET: $syslog2 = getListValue(NetworkServers,Server,WS2,IPAddr,null)
```

Above, an `Action-Command` is called by the script if the value in the variable `$location` is equal to "West." Then, the script declares two SET variables, which use the `getListValue( )` to fetch specific values from the `NetworkServers` list. As an example, that list has three data columns: `Location`, `Server` and `IPAddr`.

Location	Server	IPAddr
West	WS1	220.11.113.246
Southwest	WS3	210.15.200.246
Asiapac	AS1	132.55.210.246
Northeast	NS1	212.46.155.246
East	NS2	212.46.156.246
Midwest	MW1	210.15.205.246
Southeast	NS3	212.46.154.246
Midwest	MW2	210.15.206.146
West	WS2	220.11.114.246
South	SS1	180.99.99.246
EMEA	EM1	133.10.1.246

The variable `$syslog1` is written with the value `220.11.113.246`, because the script is told to open the list and search for the value `WS1` in the list's column labeled `Server`.

Finding `WS1` in the first row of the list, the script is directed to extract the value in that row's corresponding `IPAddr` column and write that IP value into `$syslog1`. The result is that the variable `$Syslog1 = 220.11.113.246`.

The second variable $syslog2 works in similar fashion, getting written with the value 220.11.114.246. When the script matches WS2 in the list, the script is told to extract the value in that row's corresponding IPAddr column and write that into $syslog2.

Note: For all list operations, should you need to access a list as part of script operation, make sure that all changes to the list are saved (adding and deleting rows, changing column header names or moving columns, adding new data columns) before attempting to access the list again.

Another example, showing an entire script that updates the host names of devices from a list using old and new host names:

```
Script-Filter:
   $Vendor eq "Cisco"
########################

Action:
   Get Device

Action-Description:
   Get the old Host Name from the list and replace it with the new one.

Action-Commands:
   SET: $new = getListValue(my_newhostname,old_name,$name,newhostname,NOTFOUND)

Action-Commands:{$new ne “NOTFOUND”}
   config t
   hostname $new
   end
   wr mem
```

The list my_newhostname could be of any length; the example here is quite brief:

old_name	newhostname
Wan-Router	Wan-Router2
router-10-66-20-66	EMEA_gateway
router-10-66-20-225	Asiapac_gateway

PRINT

The PRINT keyword allows the printing of simple text strings (similar to the C “printf” command) and the printing of values within variables in CCS scripts to output text files. Action-Command attributes and Trigger-Command attributes may use the PRINT keyword. You can view file output in the **Files** tab in the Job Viewer. You can append successive PRINT outputs to the same file. You can specify variables output using the $‹variable_name› directive. Avoid using other special characters as they may conflict with tokens used by the CCS scripting engine. You can use the following special characters in the file names: [a-zA-Z0-9_-].

Examples

```
PRINT: This is the output of the show running-config command
```

The statement in quotation marks is written to an automatically-named file ‹device_id›-1.log.

You can also direct the output of the PRINT keyword to a specific filename:

```
PRINT (config_listing.txt): This is the output of the show running-config command
```

The statement in quotation marks is written to a specified file named config_listing.txt.

Similar to the ARCHIVE keyword, you can use variables to dynamically specify file names:

```
PRINT ($device_devicename.txt): This is the output of the show running-config command
```

The statement writes a series of files to specified filenames for each of the devices against which the script executes, after the script-filter is applied.

PRINT can be used to append successive notes into the same output file:

```
Action-Commands:
PRINT ($device_devicename.txt): This is the output of show running-config
ARCHIVE ($device_devicename.txt): show running-config
PRINT ($device_devicename.txt): Expect this to be the same as show startup-config
ARCHIVE ($device_devicename.txt): show startup-config
```

In this case ‹$device_devicename›.txt files contain the text `This is the output of show running-config` followed by the output of an executed **show running-config** command, followed by another text statement `Expect this to be the same as show startup-config` followed by the output of a **show startup-config** command.

You can use PRINT to output the contents of variables to an external file:

```
Action-Commands:
PRINT ($device_devicename.txt): $device_devicename is a $model
```

This script directive prints the contents of two variables to the file ‹$device_devicename›.txt.

SKIPERROR

The `SKIPERROR` directive turns off error handling for script attributes when an error may appear from the acted-upon device, potentially preventing further job execution. You may issue `SKIPERROR: on` for one part of a script, thereby disabling error handling, and re-enable error handling again by issuing a second `SKIPERROR: off` directive. `SKIPERROR` can be used in `Action-Command` attributes and `Trigger-Command` attributes.

Example

```
# Turn off error handling
SKIPERROR: on
<further script execution here>

# Turn error handling back on
SKIPERROR: off
```

SLEEP

The `SLEEP` directive pauses script execution for a specified number of seconds. `SLEEP` can be used in `Action-Command` attributes and `Trigger-Command` attributes.

Example

```
# Sleep for one minute
sleep: 60
```

Commenting CCS Code

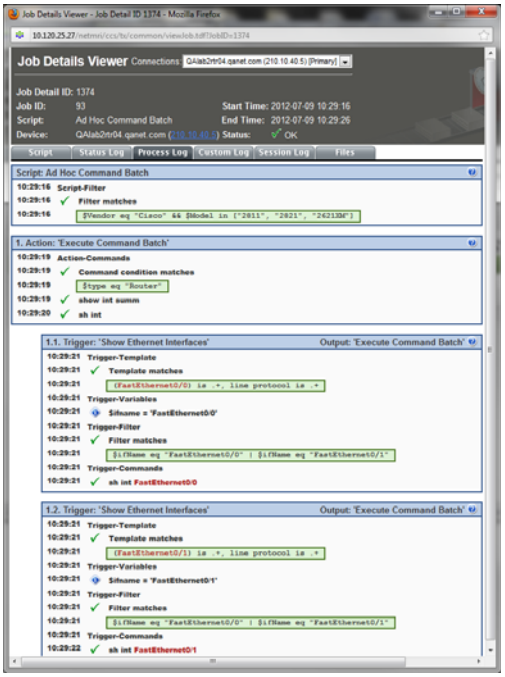
To ensure your scripts are maintainable, we recommend the liberal use of comments. All comments in CCS scripts are denoted using the hashtag (#) symbol as the first character in a line. An example:

```
#-------------------------------------------------------------------------
#  The $ifName value extracted by the first trigger will be used in this command.
#  Variables are globally scoped and referenced by any part of the script as needed.
#-------------------------------------------------------------------------
```

Multi-line comments can contain any text and are treated as comments within /* and */ delimiters, as follows:

```
/*
This is a multi line comment. Use it to explain all of the
glorious details of the script you just created.
*/
```

Looping with CCS Scripting



As previously noted, `Trigger-Commands` are the looping mechanism for CCS scripts, though simple IF-THEN logical constructs can be defined in other attributes. The sample script in this section uses two simple iterations of automated Cisco command line operations to illustrate nested loops in CCS.

You choose the device, devices or device group against which the script runs. Once that's done, the script executes beginning with the `script-filter`, which filters out all devices in the chosen device group except for specific router model types.

```
Script-Filter:

$Vendor eq "Cisco" && $Model in ["2811", "2821",
"2621XM"]
```

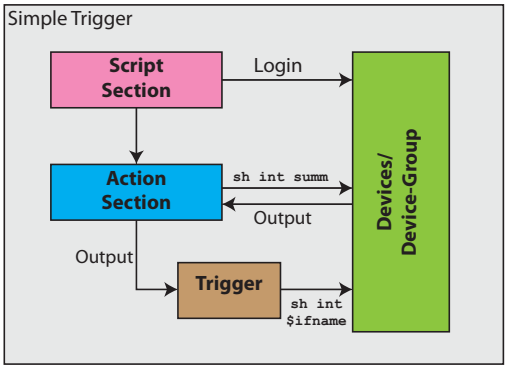
The single Action section executes a simple loop of Cisco CLI commands against every device that matches the `Script-Filter`. The output is contained in a memory buffer, log messages written and the output is also written to a file using the ARCHIVE command. Note that the Log and ARCHIVE directives do not appear in the Process Log.

```
Action-Commands: { $type eq "Router" }

    ARCHIVE ($ipaddress.txt): show int summ

    LOG-INFO: Router's interface list has been
written to file

    sh int

    LOG-INFO: issued another sh int command for full interface config

Output-Triggers:

    Show Ethernet Interfaces
```

Then, the `Output-Trigger` is called for each matching device. This represents the nested loop, which starts by establishing a `Trigger-Template` and a `Trigger-Variable`, which uses a regular expression. The trigger matches the template against the output from the **sh int** command. For matching devices, at least one interface will match the template. In the first trigger execution, a single template match appears, showing `FastEthernet0/0` in red. Then, using a `Trigger-Command` attribute, the script issues a `sh int $ifName` command (note the use of the variable) and appends the response output into the associated text file.

```
Trigger:

    Show Ethernet Interfaces

Trigger-Variables:
```

```
    $ifName   "/FastEthernet[^ ]*/"
Trigger-Template:
    [[$ifName]] is .+, line protocol is .+
Trigger-Filter:
    $ifName eq "FastEthernet0/0" | $ifName eq "FastEthernet0/1"
Trigger-Commands:
    ARCHIVE  ($ipaddress.txt): sh int $ifName
    LOG-INFO: Router's Ethernet port configs have been written to file
```



As shown in the Job Details Viewer example, a second iteration of the loop executes for the `sh int $ifName` command on the router (because the router has two Ethernet ports) and the text output is appended to the same text file. The output from the original **sh int summ** command and from the **two sh int FastEthernet** commands is concatenated to the same file. The trigger executes at least once for any router device whose Ethernet interface CLI name resembles `FastEthernet[^ ]*` and in this case executes twice.

It's a simple matter, on a Catalyst switch with 24 or 48 ports, and a slightly modified script, to automatically execute the loop 24 or 48 times.

First, change the `Script-Filter` specifications:

```
Script-Filter:
$Vendor eq "Cisco" && $Model in ["catalyst295024"]
```

Note that all arguments are case-sensitive. In this case, if you wrote "`Catalyst295024`" the script would skip all switch devices this it was intended to run against, because `$Model` wouldn't find a match.

```
Action-Commands: { $type eq "Switch" }
```

Then, to enforce the loop automatically, simply remove the `Trigger-Filter`, because you don't need it given that all ports on the device type are `FastEthernet0/*` (also, you may have either a `Trigger-Filter` or a `Trigger-Template` or both, depending on the script; you must have at least one of the two):

```
# Trigger-Filter:
# $ifName eq "FastEthernet0/0" | $ifName eq "FastEthernet0/1"
```

The same **sh int** commands execute for every port on the chosen devices; the Status Logs and Process Logs, and the written text file, are substantially longer. For a 24-port Catalyst, each device writes out a file nearly 700 lines in length.

Using Filtering on Scripted Commands

A recommended practice for trigger filtering is to avoid filtering against an entire Trigger. Filtering is more effective when you perform it against individual commands issued by Action sections.

Well-Known Variables

Among the well-known variables that can be called in any CCS (or Perl) script include the following (sample values enclosed in quotemarks):

```
$assurance = "99";
$batch_id = "262"
```

(Returns the **Job ID** value that appears in the Job History UI.)

```
$community = "mysnmp";
$contextname = "";
$device_id = "21";
$ipaddress = "220.10.110.5";
```

```
$model = "871";
$name = "QAlabrtr11";
$syscontact = "QAEast's lab; Floor 2";
$sysdescr = "Cisco IOS Software, C870 Software (C870-ADVIPSERVICESK9-M), Version
12.3(8)YI2, RELEASE SOFTWARE (fc1)  Synched to technology version 12.3(10.3)T2
Technical Support: http://www.cisco.com/techsupport  Copyright (c) 1986-2005 by Cisco
Systems, Inc.  Compil";
$syslocation = "QA lab right bottom soho router";
$sysname = "QAlabrtr11.qanet.com";
$type = "Router";
$vendor = "Cisco";
$version = "12.3(8)YI2";
```

Scripting Example

This section provides a complete script example that executes a Cisco Connectivity Fault Management feature update job on a series of Cisco 3400 switches. Note the extended series of configuration commands in the trigger process-CFM.

```
##############################################################################
## Export of Script: CFM Update
## Script-Level: 2
## Script-Category:
##############################################################################
Script:
   CFM Update
Script-Description: A script to dynamically configure CFM on switches
Script-Filter:
$Vendor eq "Cisco" and $Model like /3400/
#######
Action:
Determine CFM
Action-Description:
Do a show CFM to determine if a device already has CFM.
Action-Commands:
SET: $cfm_exists = "yes"
show ethernet cfm maintenance-points local
Output-Triggers:
Check CFM
########
Trigger:
Check CFM
Trigger-Description:
If CFM is already enabled on the device, we will update the cfm_exists variable to yes
Trigger-Template:
Local MEPs\: None
```

```
Trigger-Commands:
SET: $cfm_exists = "no"
########
Action:
Get CFM Variables
Action-Description:
Perform a "show run interface Gi0/1" to get the existing VLAN information.
Action-Filter:
$cfm_exists eq "no"
Action-Commands:
sho run interface Gi0/1
SET: $correct_vlan = "no"
SET: $UpdateMade = "no"
Output-Triggers:
Process CFM
Incorrect VLAN
########
Trigger:
Process CFM
Trigger-Description: Parse the output of the show run to get the vlan numbers.
Trigger-Variables:
$vlan integer
Trigger-Template:
switchport trunk allowed vlan 2[[$vlan]],3\d+
Trigger-Commands:
SET: $correct_vlan = "yes"
SET: $UpdateMade = "yes"
config t
ethernet cfm ieee
ethernet cfm global
ethernet cfm traceroute cache
ethernet cfm traceroute cache size 200
ethernet cfm traceroute cache hold-time 60
ethernet cfm mip filter
ethernet cfm alarm notification all
ethernet cfm domain Slight level 4
service vlan-id 2$vlan vlan 2$vlan
mep mpid 6$vlan
continuity-check
continuity-check interval 1s
continuity-check loss-threshold 2
mip auto-create
service vlan-id 3$vlan vlan 3$vlan
mep mpid 7$vlan
continuity-check
```

```
continuity-check interval 1s
continuity-check loss-threshold 2
mip auto-create
ethernet cfm lck link-status global
disable
ethernet cfm ais link-status global
disable
ethernet evc 2$vlan
oam protocol cfm svlan 2$vlan domain Slight
ethernet evc 3$vlan
oam protocol cfm svlan 3$vlan domain Slight
Int g0/1
No service-policy input CoS
No service-policy output parent-shape-1Gb/s
No l2protocol-tunnel
exit
No policy-map parent-shape-1Gb/s
No policy-map child-shape-100Mb/s
No policy-map CoS
No policy-map TXQ
policy-map child-shape-100Mb/s
class class-default
shape average 100000000
policy-map parent-shape-1Gb/s
class class-default
shape average 1000000000
service-policy child-shape-100Mb/s
policy-map 100Mb/s
class class-default
trigger-commands:{$Version like /12\.2\(55\)/}
police cir 100m conform-action transmit exceed-action drop
trigger-commands:{$Version not like /12\.2\(55\)/}
police cir 100m pir 100m conform-action transmit exceed-action drop violate-action drop
trigger-commands:
Int g0/1
Service-policy input 100Mb/s
Service-policy output parent-shape-1Gb/s
ethernet cfm mep domain Slight mpid 3$vlan vlan 3$vlan
cos 7
ethernet cfm mep domain Slight mpid 2$vlan vlan 2$vlan
cos 7
no ethernet cfm ais link-status
########
Action:
End and Write Memory
```

```
Action-Description:
End and Write Memory only if we entered config mode.
Action-Commands:{$UpdateMade eq "yes"}
end
DEBUG:write mem
########
Issue:
   Incorrect VLAN
Issue-ID:
   Incorrect_VLAN_CFM
Issue-Severity:
   error
Issue-Description:
   This device could not be upated by the CFM script
Issue-Filter:
   $correct_vlan eq "no"
Issue-Details:
   Host $Name
   Address $IPAddress
```