Dew Technology test frame work and How to write the automation case

Dew Software Technology Company Limited

2019.1.12

Catalogue

Goal

In Dew Technology's whole test framework design, there should be three parts, system level pressure and performance testing, function and regression testing (or we call function and feature testing), and the unit/API level testing (we will use the Jenkins frame work and continuous regression mode in sprint mode).
This document focus on the function and regression part, describing how the test harness works from the point of view of a programmer writing test cases or library procedures. In the future, we can also merge the GUI testing tool from third party into this framework. The testing result can be reviewed from a web also.
The goal is to provide a training document that will enable testers to both use the automated libraries and/or test cases and contribute to them.
This framework can also help to install and upgrade the Dew Technology distribution cluster in a robust and elegant way.
The commands sending out from this frame work also consider compatible in two modes: CLI and Web NMS (for future, need a third party tool to manage the web NMS and provide the interface/lib to our framework).

Versions

Revision	Who	Date	Description
0.1	QA team	1/22/2019	Initial Release

Writing test cases

Test case philosophy

A testcase is a standalone entity that is executed immediately after its parent setup procs have run successfully. Without regard to the underlying user interface method, a testcase performs **temporary modifications** to the configuration of the testbed for the purpose of verifying a simple behavior of the system under test. The changes made by a testcase are undone by the testcase before it exits, regardless of the outcome of the test.

Library file structure

Figure 1 illustrates the more prominent directories in the dewtest harness structure. When running the dewtest tool, you should be "pathed" to the lib directory, where dewtest lives. Your testbed's elements file is found under the testbed_configs directory and is a TCL file. The test_suites directory is the top level folder of the test suites you are about to run. Refer to the sample execution line, shown below, to understand how the various files play into the execution of the Regression Test Suite using a sample dewtest harness execution command (from the DEWtest/lib directory):

```
./dewtest.tcl -s "/default/Regression/HADOOP /default/ixia" -e
testbed_configs/testcluster01-elements-paraFS.tcl -m tester@dew.com -l
/var/www/html -r http://192.168.1.19/results -x '$info(runlevel) <= 4'
```

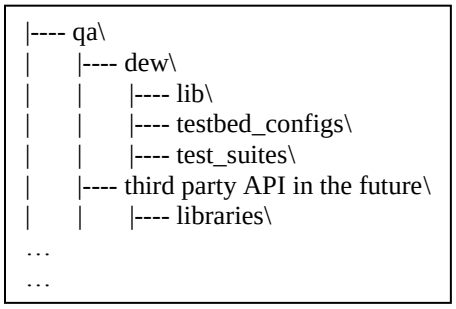
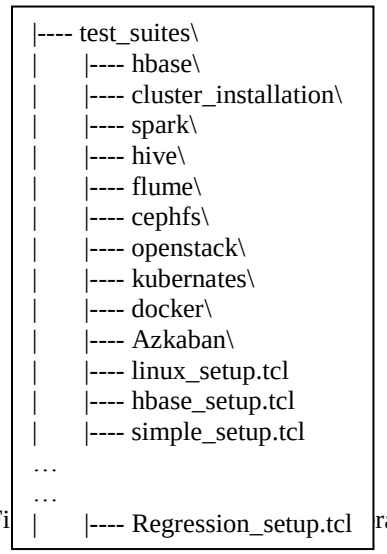


Figure 1: The test harness directory hierarchy

In the execution of the test suites, the first file of interest is **the test_suite.map** file found under the **testbed_configs directory**. This TCL file maps out the Setup, Test Case and Teardowns for the entire suite of supported test cases. When the dewtest harness is run with the -s option, the tester is specifying what suites of tests to run, as described in this file. You can see this as the first argument in the execution line above. Also, note that the suites described as arguments to the -s option are not actual filesystem paths, rather, are paths in relation to the suites as described in the test_suite.map file. Careful analysis of the map file reveals how the above path can be determined as a hierarchy.

Further examination of the test_suites folder will reveal that test cases are physically located in subdirectories to the test_suites folder. See figure 2, below.



Fi rarchy

What is important to know about his hierarchy is that it is nothing more than a way to organize and classify test cases. The cephps test cases obviously belong in the cephfs directory, etc. When the test creator needs another folder, it should be created with a related name and should be used to organize related test cases.

Also of note, in Figure 2, is the presence of the setup files. In this folder, there setup and teardown scripts live.

When the tester pulls the entire suite of tests out of SVN, they should first run:

dewtest -I

To index the entire suite and the rest of the test harness files. This provides dewtest with the means of finding any and all test cases and library functions that are executed.

Structure of a test case

Launching the execution of test cases

What follows is a slightly more detailed description of the various options to dewtest than is printed by the --help option.

Usage: **./dewtest.tcl** <options ... >

-C, --nocleanup : Skip cleanup calls

Skip running cleanup procs. Can speed up debugging of individual test cases.

-G, --casefile **<string>** : File containing list of test cases to run

A file containing a set of testcases to be run. See the doc/example_casefile for more info on the format of the file.

-H, --testdir **<string>** : Specify alternate DEWtest lib directory

By default, Dewtest looks for lib, testbed_elements, and test_suites directories in the parent directory of the dewtest executable. If the location of dewtest has moved, this option tells it where the rest of the files are located.

-I, --index : Rebuild the tclIndex for the tree

Dewtest makes extensive use of TCL function autoloading to avoid having to always source in all of the DEWtest code. The autoloader relies on having an external index of the locations of autoloadable procs. This option tells dewtest to rebuild that index. The index has to be rebuilt every time a new proc is added, or when a fresh code tree is checked out.

-J, --noobjects : Skip connection to device objects

It can sometimes take a while for dewtest to connect (or fail to connect) to the linux (or more entities in the future) objects that are defined in the testbed. That can be annoying if the objects aren't needed, like when debugging a test utility proc in interactive mode. This option disables connecting up the objects.

-L, --listlevels : List the log messgae levels.

This option lists all of the log level keywords in order of increasing verbosity. These are the keywords that can be used by the --breaklevel and --debuglevel options.

-M, --title <string> : Title of test report

By default, Dewtest titles its reports based on the hardware type and software version of the mx1 object, if it is available. This option is used to override that title. An example might be -M "Re-test of AAA on MX-8 under Bobs private image"

-O, --nosuites : Skip subsuites.

Dewtest normally runs all subsuites of a suite that has been scheduled to run. This option overrides that behavior. The testcases in a specified suite will be run, but not any of the cases that are in subsuites of the suite.

-S, --nosetup : Skip initial setup calls.

Skip running setup procs. Can speed up debugging of individual test cases.

-T, --notests : Skip test cases.

Skip running test procs. Can speed up debugging of setup and cleanup procs.

-X, --setupfilter <string> : Setup proc filter string
-x, --filter <string> : Testcase filter string

Provides a filter expression to be evaluated against each test to determine if the test should be run. The <string> is a TCL expression. The data for comparison is what the testcase has provided in the info array. You must be careful with what is passed in, because there is no protection from executing malicious code. If you pass in a string that contains a runnable expression, well... it gets run. Also be aware that most shells will try and substitute out variables before running any programs. Be sure to quote carefully! An example of a filter string might be '$info(Author) == "bob"' to indicate that only tests written by bob should be run. A more complex example might be '[string match -nocase "modo*" $info(Test_Case_Short_Description] ||
$info(runlevel) > 5' which would only select tests with 'modo' in the description or had a runlevel greater than 5.

The two options differ in that --setupfilter is only run against setup procedures, while --filter is only run against test procedures.

-a, --startat <string> : Skip test cases up to this one.

If a dewtest run fails or is stopped for any reason, it can be resumed at the last case it was working on by specifying the case name with this option. All cases that come before this one in the schedule will be skipped.

-b, --breaklevel <string> : Log message break level

Tells dewtest to enter the TestTcl debugger whenever it logs a message of the specified type.

-d, --debuglevel <string> : Debug log level

Tells dewtest to surpress all log messages below the priority of the specified type. Default is STATUS.

-e, --elementfile <string> : Specify alternate testbed element file

The element file is usually taken from **DEWtest/testbed_configs/<hostname>_elements.tcl**, where hostname is the short name of the system dewtest is running on. This option overrides that filename.

-f, --suitefile <string> : Specify alternate suite definition file

The suite file is usually taken from **DEWtest/testbed_configs/suite-map.tcl**. This option overrides that filename.

-g, --caselist <string> : List of test case procs to run

Provides a list of specific test cases to run. Remember to enclose the list in quotes. Example -g "Cephps-directory-FUNC-001b Hbase-create_table-004c Hive-delete-Negative-003". If running more than a few test cases individually, have a look at the -G, --casefile option.

-h, --help : Display this message

Pulls up the list of dewtest command options.

-i, --interactive : Interactive mode.

Instructs dewtest to enter the interactive debugger instead of building and running a test schedule.

-l, --logdir **<string>** : Specify alternate log directory

By default, dewtest creates a results directory in the test directory (the DEWtest directory). This option allows the user to specify a different location for the results directory. example: --logdir /tmp

-m, --mailto **<string>** : Email address of text report recipient

Dewtest can mail its final report out to a list of recipients, specified as a
list using this option. Remember to enclose the list of recipients in quotes.
Example: --mailto "jeff@Dew Technology.com bob@Dew Technology.com sqa@Dew Technology.com"

-r, --reporturl **<string>** : URL path to logdir

At the end of a report, dewtest included a URL to the location of the results file. By default, that is a file:// URL, which is only valid on the local machine. This option is used to turn it into an http:// url by providing dewtest with the correct html document root for the location of the logs.
Example: --reporturl http://192.168.1.99 /~bob/DEWtest/results

-s, --suitelist **<string>** : List of suites to run

Provides a list of suites to run. Suites are specified like unix filename paths, and must contain the full path to the suite to be run. Example:
--suitelist "/default/Regression/Hbase/tables/insert
/default/Regression/Hadoop/BIGCFG"

-t, --test : Test schedule, same as -C -J -S -T

This option is for testing what cases will be scheduled to run without actually running any of them. It is used to check that a run will contain all of the test cases that you expect it to. It is exactly the same as specifying the -C -J -S -T options, just less to type.

-u, --ui_pref **<string>** : User interface preference (CLI or NMS)

Determines the user interface preference for the run. If it is NMS, dewtest autolib commands will try to use web interface (in future) whenever possible. When it is CLI, the commands will be attempted using the linux or unix command line interface.

Adding test cases to the harness

There are 3 steps that must be followed to add a new testcase.....

1) Write the testcase and place it somewhere under DEWtest/test_suites
2) Add the testcase to a suite_map file
3) run "dewtest.tcl -I"

Step 1 - Write the testcase and place it under DEWtest/test_suites
---
All testcases should be placed in one of the directories **under DEWtest/test_suites**. There are directories for Cephps, DEWtestbase, Sif and others. Simply determine which directory most closely matches the description of the testcase you're adding and place your testcase there. If none of the existing directories seem right, create a new directory under DEWtest/test_suites with an appropriate name. Make sure you do a "cvs add/svn update" of the new directory as well as the testcase.

Step 2 - Add the testcase to a suite_map file

All testcases to be executed must be listed in a suite_map file. By default, dewtest uses the file DEWtest/testbed_configs/suite_map.tcl. This is the file used for regression testing. Other suite_map files can also be created for other forms of testing or for private tests. The structure of a suite_map file is a heirarchical list of testcases and setup files. A fragment of DEWtest/testbed_configs/suite_map.tcl is shown below....

```
suite default {
   setup default_setup
   suite Regression {
      setup Regression_setup
      suite Hadoop {
         suite ParaFS {
            setup configuration_setup
            testcase Hadoop-parafs-FUNC-001
            suite Parabase {
               testcase GP-ADMIN-FUNC-005
               testcase GP-APP-FUNC-004
            }
            suite Webclusteradd {
               setup Webcluster_setup
               testcase cluster-WEB-FUNC-001a
               testcase cluster-WEB-FUNC-002
               testcase cluster-WEB-FUNC-003
            }
.
.
```

Notice that testcases are grouped according to function and name. For example, all WebAAA tests are located in the suite /default/Regression/Hadoop/Cephps/WebAAA. **This grouping allows similar testcases to share a common setup file** (clusterWeb_setup in this case), and allows the person running tests to select a subset of tests to run (e.g. dewtest.tcl -s /default/Regression/Hadoop/Cephps/WebAAA will run all WebAAA tests). Any test that should run as part of NOS regression should be placed somewhere under /default/Regression/NOS. Similary, any test that should be run as part of Web regression should be placed somewhere under /default/Regression/Web.

Note that there isn't a one-to-one mapping between the directories from step 1 above and the location in the suite-map file.

Step 3 - run "dewtest.tcl -I"

Once a testcase has been added to a suite_map file, dewtest will need to know where the testcase is located in order to run it. The command "dewtest.tcl -I" causes dewtest to re-**generate the TCL index files** that tell it the location of each procedure and testcase. This only needs to be done once each time a new testcase is added. If the user forgets to execute this command, dewtest will print out an error about not being able to find the new procedure/testcase.

Test case Do's and Don'ts

* DO assume that when a testcase is run, all of the test setup procs leading up to it have run successfully, and no other testcase have run. The environment presented to a testcase is clean; there is no need to double check that.

* DO fill in all of the informational variables in the info block of the testcase. The values are printed out as part of the final testcase report, and help other people understand what the test is intended to do.

* DO have your testcase perform everything within its power to clean up after itself. This is done for execution speed. The test harness can clean up after a testcase if it must, but the harness takes a long time to do so-- it is almost always faster for a testcase to undo its small number of changes than it is for the harness to reload from the default configuration and rerun all of the setup procs.

* DO NOT set the info(cleanedup) variable to true at the start of the testcase.
Wait to set it until the just before the case returns. The harness trusts that if info(cleanedup) was set to true, the case really did mop up after itself.
Don't violate that trust by setting it to true at the start and changing it to false only if a cleanup failure is detected. Your case might bail out unexpectedly without setting the value back to false, and the harness might be watching that variable as your testcase executes.

* DO write your testcases so that they can run on any testbed that supplies the necessary information in the elements file.

* DO NOT hard code values into your testcases. Environment specific values must go into the testbed_elements file. This includes, but is not limited to: port assignments, IP addresses, platform specific limits, fingerprints, external servers such as RADIUS or NTP, VLAN tags, and timing DEWtestmeters.

* DO NOT load saved CLI configurations onto MX's as part of a test. It is not portable across testbeds.

* DO NOT write testcases that depend on other test cases having been run first. Keep it independent always.
The only assumption that you can make about the run order of test cases is that the setup procs will have run before this testcase, and that some other testcase may run after it. The test harness may or may not reset the configuration environment after a testcase has run, and there is no way for a testcase to control that. You can't use a testcase to set something up for another testcase. You must use a setup proc for that.

* DO as much shared configuration as possible within the setup procedures. A good rule of thumb is that if more than two testcases have the same setup requirements, they should be put into a sub-suite so that the setup proc can be written once and shared. Doing so will simplify the test cases, and will help speed up the whole regression run.

* DO NOT assume that your testcase is running under a particular UI method.
The LinuxAutoLib commands are there to hide the UI from you so that you don't have to re-write the testcase for each UI Method. If you write code that relies on a specific UI method, it may fail when run against another UI method outside of your particular testbed.

* DO feel free to use the set_testbed_aps proc inside of your testcase instead of a direct module reference within the elements file. The set_testbed_aps proc searches the set of elemetn's defined in the elements file for ones that match exactly the DEWtestmeters that your test requires.

Using the proc helps keep thenumber of testcase specific lines in the elements file under control. Check out the proc definition for an example of how to use it.

* DO feel free to add to the suite_map file.

* DO feel free to add to the testbed_elements file.

* DO NOT go crazy adding elements to the testbed_elements file. Remember, the more values in that file, the more has to be changed when running a test in a different environment. Add them when you need to, but remember there is a cost associated with doing so.

* DO add useful non-LinuxAutoLib utility procs to lib/test_utils.tcl. Use the rule of threes-- if you needed it three times, it is worth making a utility proc out of. Even if you've only needed it twice, odds are good that someone else will need it that third time.

Using the TCL debugger

Dew/dewtest uses a subset of the TestTcl package for an interactive debugger. It is a very simple debugger in that it only provides four commands, but it is a powerful tool because it provides a full tcl interpreter inside the context of a running program. The debugger is usually used in two ways, it can be invoked by dewtest for problem debugging whenever a log message of a certain type is generated (for example an ERROR), or it can be used as an environment for interactive testcase development.

TestTcl Built In Commands

When the debugger begins, it outputs a list of its built-in commands:

```
**** Test-Tcl Debugger
**** Builtin commands are bt, up, down.
**** Type Ctrl-D to resume.
Raindrop>
```

The built-in commands are used to view and walk the TCL execution stack.
The stack is the chain of function calls leading up to the place where the debugger was invoked. The debugger can execute commands at any stack level.

To view the current stack, use the **bt** command. Here is an example using the go_interactive testcase provided by the null_testcase.tcl example in the doc directory:

```
dogboy:~/localhome/sw/qa/para> lib/dewtest.tcl -f testbed_configs/dogboy-
suite-map.tcl
1 executable cases found.
Logging to: /usr/local/home/jjahr/sw/qa/para/results/Sep07-22h54m45s
22:54:50 TITLE: go_interactive
22:54:50 PURPOSE:
22:54:50 STEP: Ok, yer interactive now.
**** Test-Tcl Debugger
**** Builtin commands are bt, up, down.
**** Type Ctrl-D to resume.
DoSomething> bt
   1 - main {-f testbed_configs/dogboy-suite-map.tcl}
   2 - run_case {/default go_interactive} info 0 0 0
   3 - wrap_case go_interactive run info
   4 - go_interactive run info
   5 - TestTcl::breakpoint DoSomething

Command execution level is 4.
DoSomething>
```

This shows that the debugger was invoked from within a proc called go_interactive that was called with the arguments {run info}. In this example, go_interactive is the name of test case-- all it really does is to call the debugger and return a pass when it exits. The stack shows the full set of calls within dewtest that led up to the testcase being called. Any TCL commands that are entered into the debugger act as if they were inside the source code of the proc that is at the debuggers current command execution level. At this point in the example, the debugger is at level 4, inside the go_interactive proc, so the TCL command 'info locals' will show the names of all the variables local to the go_interactive proc.

```
DoSomething> info locals
command infoarray args this_proc cleanup_proc ret
DoSomething>
```

It is also possible to view and modify the values of these variables 'on the fly'. For example, the go_interactive testcase tracks its return value in the variable named ret. We can force go_interactive to fail when the debugger exits by changing the value of ret:

```
DoSomething> set ret
true {}
DoSomething> set ret [list false "Because I said so"]
false {Because I said so}
DoSomething>
```

The up and down commands change the current execution level do a different point in the stack. This is useful if the testcase being debugged has subroutines. In this case, going 'up' enters the wrap_case proc, which is part of the test harness. There isn't much there to look at, but notice that the command 'info locals' now shows the local variables for wrap_case instead of go_interactive:

```
DoSomething> up
Command execution level is 3.
DoSomething> info locals
procname command infoarray args
DoSomething>
```

TestTcl for Problem Debugging

Dewtest provides an option, -b, or --breaklevel, for dropping into the TestTcl debugger whenever a log message of a certain type is generated.

For example, to have the test harness pause in the debugger anytime that a testcase logs an ERROR message, pass the --breaklevel ERROR option to dewtest. It is also sometimes useful to stop a case at critial points to verify that the code is performing as expected. This is can be accomplished by including write_debug statements in the test cases, and running under --breaklevel DEBUG.

TestTcl for Interactive Developement

Dewtest provides two ways to do interactive development. The first is the -i, or --interactive flag. When run with that option, dewtest drops into the debugger just after setting up the testcase environment, before it would normally have started executing any test cases or setup procs.

This mode is useful for experimenting with Linux or module objects, or just as a TCL shell that has access to all of the DEW libraries. For example:

```
dogboy:~/localhome/sw/qa/para> lib/dewtest.tcl -i
No runnable cases found.
Logging to: /usr/local/home/jjahr/sw/qa/para/results/June18-23h25m59s
23:26:03 INFO: Linux objects are: Spark Zookeeper
23:26:03 INFO: Linux objects are:
23:26:03 INFO: **** Entering interactive mode ****
**** Test-Tcl Debugger
**** Builtin commands are bt, up, down.
**** Type Ctrl-D to resume.
hcl>
```

The second method for interactive development is to start with a testcase template that is empty except for a single write_debug statement. Insert the testcase into the suite map so that the intended setup procs will be called before the testcase is run, then have dewtest execute that single testcase using the -g, --caselist option and --breaklevel DEBUG. The system will then enter the TestTcl debugger just after setting up the environment that the testcase will be expected to run in. At that point, it is possible to configure the system 'live' using LinuxAutoLib commands, and copy the working code fragments directly into the empty testcase template.

This method of interactive development is useful because it allows you to experiment with the function calls on the spot and verify exactly how they function. For example, the LinuxAutoLib commands can all be called with a -help option. That may not make sense to do inside of a program, but it is very handy when inside the debugger:

```
para> Linux1 LinuXGetVersion -help
true {
NAME:
     ::LinuxAutolibClass::LinuxGetVersion

LOCAL UI METHODS:
     cli

REQUIRED ARGUMENTS:

OPTIONAL ARGUMENTS:
               LinuxName = The name of the LinuX
               MODULEName = The name of the module in the machine
                 Next = If set, get next boot image instead of current }
para> Linux1 LinuxGetVersion -Next
true 4.0.9.private
para>

Because the debugger is an interactive TCL shell, it can be used to test that
code sections will work correctly.  That can save time when writing regular
expressions. For example:

para> set version [data_from [Linux1 LINUXGetVersion -Next]]
4.0.9.private
para> set pattern "(.*)(private)"
(.*)(private)
para> regexp -- $pattern $version all numbers build
1
para> set numbers
4.0.9.
para> set build
private
para> set pattern "(.*)\[.\](private)"
(.*)[.](private)
para> regexp -- $pattern $version all numbers build
1
para> set numbers
4.0.9
para>
```

By default, the interactive mode doesn't print the output of the write_debug commands on the screen. You have to use the following command to enable it:

```
LogMsg::setDesc stdout DEBUG ansi
```

You can also use:

```
LogMsg::setDesc stdout ANY ansi
```

But don't use these commands in any script.

Once the code behaves correctly in the debugger, it can be copied into a testcase with some confidence that it will work as intended.

Stupid Debugger Tricks

Calling the debugger directly:

If you ever want to call into the debugger without having to --breaklevel on a write_log message, put this command into your code at the point where the debugger should be called:

TestTcl::breakpoint MyPrompt

The argument determines what the debugger prompt will be. When you exit the debugger with Ctrl-D, your program will continue running at the point after where the debugger was called.

If you ever leave a TestTcl::breakpoint command in a testcase and unintentionally hang the automated regression with it, the penalty is a manual run of testing. The point is that it is far safer to use the -b option than it is to jump directly to the debugger, but sometimes it is a nice trick to know about.

Other TestTcl commands:

It's often the case while in the debugger that you will want to examine the contents of a TCL list or array. TestTcl has two commands for "pretty printing" lists and arrays. They are ppl and ppa. You can call them directly from the TestTcl namespace (as in the example below) or import them with {namespace import TestTcl::pp*} and leave off the TestTcl:: part. Example:

```
DoSomething> info locals
command infoarray args this_proc cleanup_proc ret
DoSomething> TestTcl::ppl [info locals]
    0 = command
    1 = infoarray
    2 = args
    3 = this_proc
    4 = cleanup_proc
    5 = ret

DoSomething> array names info
Setup Procedure end_of_life Test_Case_Short_Description required_ui_method
RESULT reason cleanedup start_of_life Pass_Criteria runlevel Purpose runnable
Test_Case_ID Reference
DoSomething> set info(start_of_life)
1.0.0
DoSomething> DoSomething> TestTcl::ppa info
    0 Setup                       =
    1 Procedure                   =
    2 end_of_life                 =
    3 Test_Case_Short_Description = Nothing yet
    4 required_ui_method          =
    5 RESULT                      =
    6 reason                      =
    7 cleanedup                   = false
    8 start_of_life               = 1.0.0
    9 Pass_Criteria               =
   10 runlevel                    = 0
   11 Purpose                     =
   12 runnable                    = true
   13 Test_Case_ID                = go_interactive
   14 Reference                   =

DoSomething> namespace import TestTcl::pp*

DoSomething> ppl [info locals]
    0 = command
    1 = infoarray
    2 = args
    3 = this_proc
    4 = cleanup_proc
    5 = ret

DoSomething>
```

Log levels

DEWtest provides a logging facility for categorizing the output of a test run. It is used to provide structure to the output, and to limit the total amount of output to a manageable level. The log messages are also the primary hook into the TestTcl debugger. Knowing how and when to use the different log messages is therefore an important part of working with DEWtest.

The log levels supported in Dewtest can be listed by running dewtest with the **-L, --listlevels**, option. Dewtest will list them in order of increasing verbosity, which is another way of saying that the list is ordered by how much output each log type is expected to generate. Some of the different log levels, like RESULTPASS, TITLE, and RESULTFAIL, are weighted the same by the logger. The upshot is that a request for logs at or above level TITLE will include messages of type RESULTPASS and RESULTFAIL, but not at level ERROR.

Here is the current list of log levels along with an explanation of each. Log levels with the same weight are grouped together, and the list is ordered the same way as dewtest outputs it.

NONE

Output nothing.

RESULTPASS

Passing test results only.

TITLE

Test case titles only.

RESULTFAIL

Failing test results only.

ERROR

Any error encountered within a testcase.

UNKNOWN

Used when logmsg is called with an invalid level.

WARNING

Any non-critical, but unexpected or undesirable behavior encountered while running a testcase.

STEP

A generic testcase step, corresponding roughly to an action listed within the testplan being automated. An example of a step might be "start a process". Steps are often followed by INFO messages describing the details of how the step is being accomplished.

PURPOSE

The purpose of a testcase.

INFO

Information derived during execution of a testcase that should be shown to the operator, such as how long an action took to execute, or what device was chosen for particular test step.

STATUS

Provides operational status when an activity might take a long time to execute. An example might be, "Waiting ten more seconds for user login."

DEBUG
Testcase level debugging statements.

PUTS
Indicates accidental use of puts call within a testcase. Testcases should use a write_log call instead of puts.

CLIDEBUG
Raw CLI output to and from the devices under test.

LIBSTATUS
Provides operational status within an AutoLib library call.

LIBERROR
Any error found within a library call. The library is still expected to return the proper error response.

LIBWARNING
Any warning generated from within an AutoLib library call.

LIBDEBUG
AutoLib library debugging statements.

ANY
Any message at all.

Generating a log messages

For clarity and compatibility with our previous regression system, DEWTEST provides a set of procs named write_<level> for sending a message to the log. These are the write_ commands that can be called from within a testcase:

write_status
write_step
write_info
write_error
write_warning
write_debug

These are the ones that can be called from within AutoLib:

write_liberror
write_libwarning
write_libstatus
write_libdebug

Don't try using testcase write the _* commands from within a library, or write_lib* commands from within a testcase. Also, **don't call logMsg directly** in a script.

Turning Color Logs on and off

The console output style is controlled by the LOGMSG_STYLE environment variable. If LOGMSG_STYLE is not set, or is set to "text", no color information will be output. If it is set to "ansi", then ANSI color control codes will be sent to the terminal. You'll have to be using a terminal that supports ANSI color in order to see the difference. Color-xterm and rxvt work well under linux. Putty works well under windows.

For example, I like seeing color logs, so I have this line in my .cshrc:
setenv LOGMSG_STYLE ansi

For a cheap thrill, try setting LOGMSG_STYLE to html.

Limiting the Spew

By default, dewtest emits logs at and above the STATUS level. This can be changed by use of the -d, --debuglevel option. The option takes one argument, which is one of the log levels shown by the -L, --listlevels option. The debuglevel is used to increase or decrease the amount of output that goes to the console.

Turning down the debuglevel has the effect of sending less information to the console. That can seem like a problem when a testcase fails-- none of the debug information will be available in the console scrollback buffer! That is ok though, because when dewtest runs, it always creates a set of log files. The default location is under DEWtest/results, but you can override that with the -l, --logdir option. The set of log files are written in HTML, and correspond to the output generated at the ANY, DEBUG, and STATUS levels. When they are being written the files are line buffered, so that they always contain the most recent set of logs. You can load those files into a web browser to check status while the testcases are still running, or long after the test run has finished to go over any failed cases.

It is a good idea to go into the results directory from time to time and clear out the old test results-- dewtest won't do it for you. You may be surprised by how many old sets of results you have lying around.

Sample test case

Writing library code

Modifying files

When editing files in the test harness, use **soft tabs** (**4 spaces** per tab).
In VIM, the commands to set tabbing to 4 spaces are:

```
:set tabstop=4
:set shiftwidth=4
:set expandtab
:set softtabstop=4
```

Modify your ~/.vimrc to include these commands.

Function header template

The function headers look like this:

```
::itcl::body ::LinuxAutolibClass::ModuleCheckconfiguration { args } {

     # Set the argument options for the command here.
     set info(author) "Jeff Jahr"
     set info(created) "Fri Sep  2 16:03:48 PDT 2005"fug

     set info(description) "Returns a list containing one element per radio.
     Each element is a list of which bands are suppored on that radio."

     set info(bugs) "

     Hi! Glad you are reading this.  You must have discovered the problem
     with how this was coded, in that a data table is stored here in
     the source code, and it has to be updated every time we add a new model.
     Yep, you are in the right place to add your update, unless of course you
     want todo a better way to handle this table.

     Maybe it ought to be in the elements file?  Or some adjunct table?  Figure
     it out."

     set optional(ModuleName) "The name of the module"
     set optional(LinuxXName) "The name of the LinuxX"
     set required(Model) "The model type to check"

     # The rest is boilerplate, no need to touch.
     set cmd [lindex [info level 0] 0]
     if { $autolib_version != $libversion } {
         [return [chain $args]]
     }
     return [eval explicit_autolib $cmd required optional $args]
}
```

The help is directly integrated into the function. So when you call the help for this function you will have:

```
hcl> Linux1 ModuleCheckconfigurationSupport -help
true {
NAME:
     ::LinuxAutolibClass::ModuleCheckconfigurationSupport

LOCAL UI METHODS:
     cli nms

AUTHOR:
     Jerryf Harry

BUGS:
     Hi! Glad you are reading this. You must have discovered  the problem
     with how this was coded, in that a data table is stored here in  the
     source code, and it has to be updated every time we add a new model.
     Yep, you are in the right place to add your update, unless of course
     you  want todo a better way to handle this table.

     Maybe it ought to be in the elements file? Or some adjunct table?
     Figure  it out.


CREATED:
     Fri June 2 16:03:48 PDT 2018

DESCRIPTION:
     Returns a list containing one element per radio.  Each element is a
     list of which bands are suppored on that radio.

REQUIRED ARGUMENTS:
                Model = The model type to check

OPTIONAL ARGUMENTS:
               LinuXName = The name of the MX
               ModuleName = The name of the module in one machine }
hcl>
```

Multiplatform support

Use of **$syn(**show**)** instead of show only. This is because the ‘show’ command can be different depending on the OEMs (ex: CentOS uses ‘**disp**’ instead of ‘show’ for Fedora).

Function naming

In order to promote consistency, library functions should share a common naming convention. In general, library functions should have a name composed of a noun followed by a verb. To see a list of all library functions that have already been defined and/or implemented for Linuxs and others, use the following steps...

1) "dewtest.tcl -i" -- this starts dewtest in interactive mode.
2) From the dewtest debugger prompt, type "Linux1 help"

dewtest will return a list of all defined library functions, which will look like.....

```
hcl> Linux1 help
bad option "help": should be one of...
  linux1 ACLCopy ?arg arg ...?
  linux1 ACLDelete ?arg arg ...?
  linux1 ACLModify ?arg arg ...?
  linux1 ACLVerify ?arg arg ...?
  linux1 ACLVerifyNot ?arg arg ...?
  linux1 AccountingStatsGet ?arg arg ...?
.
.
.
```

Commonly used verbs are: **Add**, **Modify**, **Verify**, **Delete**, **VerifyNot** (verify that something does NOT exist in the configuration). With these verbs as a guide we can see that for VLANs, for example, we would define the following library functions: VLANAdd, VLANModify, VLANVerify, VLANDelete, and VLANVerifyNot. This same naming convention is used for library functions implemented both for CLI and for web management. The function naming uses the capital convention: the words are not seDEWtestted but have a capital at the beginning (ex: NetworkPlanDelete, but not Networkplandelete nor Network_Plan_Delete). Some special keywords use capital for all their letters (ex: ACL, VLAN, IGMP, RADIUS, SNMP).

Common argument names used in the LinuxAutolib Class

The following arguments are used in several LinuxAutolib class methods, so it would be consistent to reuse them:

LinuxXName	Switch name (ex: hadoop000)
ModuleName	Module name in on machine
directoryName	data directory name (ex: /opt/data)
RADIUSServerName	Name of the RADIUS server

A to Z steps to add member methods to LinuxAutolibClass

In the case a feature is not supported in the library, here are the steps to implement it. Our example is the L2 restriction feature in 4.1.

1/. Choose a **prefix** that respects the existing conventions for your feature [here: L2Restrict]

2/. You may have to implement several functions (Add, Modify, Copy, Verify, Delete, VerifyNot). We will focus on implementing the Add procedure that will be called L2RestrictAdd.

3/. **Declare** your member procs in LinuxAutolibClass.tcl:

```
public method L2RestrictAdd { args } {}
private method L2RestrictAdd_cli { args} {}
private method L2RestrictAdd_nms { args} {}
```

Note that only L2RestrictAdd is public, the 'sublayer procs' (L2RestrictAdd**_cli** and method L2RestrictAdd**_nms**) are private because L2RestrictAdd will decide to call either the CLI version or the web interface version depending on the **$ui_method** varaiable.

4/. Go to the end of the file and **define** your procedures, starting by method L2RestrictAdd:

Here there is pretty much nothing to do, just define the proc with the correct **name**, and set the **optional** and **required** array to the proper elements.

That's all, just copy the 'boilerplate' part and you're done, the **explicit_autolib** procedure will take care of argument parsing for you (using the ProcessArgs package) and will call the correct proc (either CLI or NMS version).

At this level of the function, if you specify an array called **info**, any data in this array will be printed in the help.

The common keys to fill in the info array are:

- **Author**
- **Description**
- **Bugs**
- **Examples**
- **History**

NOTA: The line LinuxName DEWtestmeter in **mandatory**. It is used internally for compatibility with the web NMS library. Web nms can either be required or optional, but it has to be there, even if the CLI function doesn't use it.

Note on the verify functions:

The philosophy is not to return at the first error. The function will continue checking what it is supposed to check and then return {false {*listing of all the problems encountered*}}.

This allows useful debugging when a function fails.

```
::itcl::body ::LinuxAutolibClass::L2RestrictAdd { args } {

    # Set the argument options for the command here.
    set optional(Mode) "Enable/Disable"
    set optional(Gateways) "List of permit gateways"
    set optional(LinuXName) "The name of the LinuX"


    set info(Author) "Tierre Cherry"
    set info(Description) "
    For a list of VLANs, modify the configuration of the L2 restrict feature"

    set info(Bugs) "MMS doesn't accept enabling this feature without
specifying at least one gateway.
    So if there is no existing gateway already configured on the selected VLAN
and you don't specify -Gateways, calling the function using -Enable will be
rejected by CLI"
    set info(Examples) "
    L2RestrictAdd -VLANName black -Mode enable -Gateways {00:ff:ff:ff:ff:fd
00:00:00:00:ff:fd}
    L2RestrictAdd -VLANName black -Mode disable
    L2RestrictAdd -VLANName {black yellow} -Mode enable"

    set info(History) "Feature appeared in WDH 4.1"

    # The rest is boilerplate, no need to touch.
    set cmd [lindex [info level 0] 0]
    if { $autolib_version != $libversion } {
        [return [chain $args]]
    }
    return [eval explicit_autolib $cmd required optional $args]
}
```

5/. Define the **CLI** version (see code on the next page).

Here is where your real coding start, sending commands to the Linux object and parsing the results.

Make sure you use 'argumentarr args' as the arguments for this function, and add the '**upvar $argumentarr arg**' command line. This allows you to get the passed arguments in the **arg** array.

For example, if you called the function like this:

```
L2RestrictAdd -VLANName black
```

Then you can get the argument in the proc in **$arg(VLANName)**. It will return 'black'.

From this point your CLI function frame is already in place: if you call the help for it (L2RestrictAdd -help), the required and optional arguments will be printed.

Now the CLI interaction starts. Usually a procedure is composed:

1 A part doing some **extra checks** that the argument parser cannot do (ex: some arguments are contradictory and shouldn't be called at the same time, or checking that VLANNumber is an integer for instance). Then it is your choice to either stop the execution of the proc providing an explicit message (return [list false "VLANNumber is not an integer"]) or issue library warnings (write_libwarning "VLANNumber is not an integer") and continue the

execution. For instance if VLANNumber is not an integer, maybe the user passed the VLAN name instead of the number so you can do the conversion using VLANName2Number.

2 A part **building the command line** that will be issued to the CLI.

Depending on the arguments used, the command issued will be different. In the following example, if the 'Mode' argument is present, then we will issue add the mode keyword to our command line, followed by the value of the argument, `$arg(Mode).`

3. A part where you **issue the command line**:

```
$this clisend "$cmd"
```

4. A part where you check if the command failed using **problem_with.**

5. A part where you **parse the result of the CLI** to extract the info needed. This part is not in the L2RestrcitAdd_cli function as here we're just setting DEWtestmeters thru the command line but would be in the L2RestrcitVerify_cli as this function requires matching the user inputs against the CLI input.

6. A **return part**.

Function always returns a list of two elements. The first one is either 'true' or 'false', the second one is the explanation ('success' or 'the command failed because of a syntax error' or most of the time it contains the error returned by the CLI that we get using the **data_from** function).

```
::itcl::body ::LinuxAutolibClass::L2RestrictAdd_cli { argumentarr args } {
    upvar $argumentarr arg

    # check combination of args makes sense
    if {![info exists arg(Mode)] && ![info exists arg(Gateways)]} {
        write_libwarning "No mode nor gateway selected: command will fail
because incomplete"
    }

    set L_VLAN "$arg(VLANName) "
    if {[info exists arg(VLANNumber)]} {append L_VLAN $arg(VLANNumber)}

    set fail 0
    set reason ""
    foreach vlan $L_VLAN {
        # setup the basic command string
        set cmd "set security l2-restrict vlan $vlan "

        if { [ info exists arg(Mode) ] } {
            write_libdebug "Using Mode $arg(Mode) on VLAN $vlan"
            append cmd "mode $arg(Mode) "
        }
        if { [ info exists arg(Gateways) ] } {
            write_libdebug "Adding permit gateways $arg(Gateways) on VLAN
$vlan"
            append cmd "permit-mac $arg(Gateways)"
        }

        set ret [ $this clisend "$cmd" ]

        if { [problem_with $ret data] } {
            incr fail
            lappend reason "{Failed on VLAN $vlan} {[data_from $ret]}"
        }
    }

    if {$fail} {
        return [list "false" $reason]
    } else {
        return [list "true" "success"]
    }
}
```

Appendix: Active Tcl installation guide

The download is a tar-gzipped archive. To extract, enter the following command at the shell prompt:
tar zxf /path/to/ActiveTcl-download.tar.gz

In the directory where you extracted the archive, run the install.sh installer script, which will automatically select between the GUI or text-based installer depending on your system configuration. After installation, make sure that the directory containing the installed executables (ActiveTcl/bin) is included in your PATH variable.

export PATH="/opt/ActiveTcl-8.6/bin:$PATH"
You can also add the man directory to your path to access the man page documentation on the command line.
export PATH="/opt/ActiveTcl-8.6/man:$PATH"

```
[root@hadoop000 lib]# rpm -Uvh epel-release*rpm
warning: epel-release-6-8.noarch.rpm: Header V3 RSA/SHA256 Signature, key ID 0608b895: NOKEY
Preparing...                ########################################### [100%]
Updating / installing...
   1:epel-release-6-8       ########################################### [100%]
[root@hadoop000 lib]# yum install tcl-tclreadline
[root@hadoop000 lib]# yum install tcl-tclreadline
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * epel: mirrors.tongji.edu.cn
Resolving Dependencies
--> Running transaction check
---> Package tcl-tclreadline.x86_64 0:2.1.0-3.el6 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package              Arch           Version                Repository      Size
================================================================================
Installing:
 tcl-tclreadline      x86_64         2.1.0-3.el6            epel            56 k

Transaction Summary
================================================================================
Install  1 Package

Total size: 56 k
Installed size: 206 k
Is this ok [y/d/N]: y
Downloading packages:
warning: /var/cache/yum/x86_64/7/epel/packages/tcl-tclreadline-2.1.0-3.el6.x86_64.rpm: Header V3 RSA/SHA256 Signature, key ID 0608b895: NOKEY
Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
Importing GPG key 0x0608B895:
 Userid     : "EPEL (6) <epel@fedoraproject.org>"
 Fingerprint: 8c3b e96a f230 9184 da5c 0dae 3b49 df2a 0608 b895
 Package    : epel-release-6-8.noarch (installed)
```

```
 From       : /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
Is this ok [y/N]: y
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Warning: RPMDB altered outside of yum.
  Installing : tcl-tclreadline-2.1.0-3.el6.x86_64                          1/1
  Verifying  : tcl-tclreadline-2.1.0-3.el6.x86_64                          1/1

Installed:
  tcl-tclreadline.x86_64 0:2.1.0-3.el6

Complete!
[root@hadoop000 lib]#
```