
# OSEHRA Certification

# %ut - A Unit Tester For M Code

**Version 1**

**August 6, 2014**

## Revision History

| Date | Revision | Description | Author |
|---|---|---|---|
| 7/31/14 | 0.1 | Draft | Joel Ivey |
| 8/6/14 | 1.0 | Initial Publication | Joel Ivey |
| 8/29/14 | 1.0 | Revision | Joel Ivey |

# Table of Contents



# Table of Figures

# %ut - A Unit Tester For M Code

## 1. Purpose

This document describes M–Unit Test, a tool that permits a series of tests to be written to address specific tags or entry points within a project and act to verify that the return results are as expected for that code. If run routinely any time that the project is modified, the tests will act to indicate whether the intended function has been modified inadvertently, or whether the modification has had unexpected effects on other functionality within the project. The set of unit tests for a project should run rapidly (usually within a matter of seconds) and with minimal disruption for developers. Another function of unit tests is that they indicate what the intended software was written to do. This can be especially useful when new developers start working with the software or a programmer returns to a project after a prolonged period. Ensuring that well-designed unit tests are created for each project, therefore, assists development, enhances maintainability and improves end-user confidence in the deployed software.

The concept of Unit Testing was already in place before Kent Beck created a tool that he used in the language Smalltalk, and then was turned into the tool Junit for Java by Kent Beck and Erich Gamma. This tool for running specific tests on facets of a software project was subsequently referred to as xUnit, since NUnit was developed for .NET developers, DUnit for Delphi developers, etc. MUnit is the equivalent tool for M developers to use and was originally created in 2003.

## 2. Scope

This document describes the use of the M–Unit Testing tools for building and running unit tests for M code. It also describes the installation of the M–Unit Test software.

## 3. Introduction to M–Unit Testing

A Unit Test framework permits small tests to be written to verify that the code under examination is doing what you expect it to do. Generally, the tests are performed on the smaller blocks of the application, and do not necessarily test all of the functionality within the application. These tests can be run frequently to validate that no errors have been introduced subsequently as changes are made in the code. The concept of automated Unit testing was introduced by Kent Beck, the creator of eXtreme Programming methodology, with a tool used in the language Smalltalk. The common JUnit framework for Java, upon which other frameworks are based, was written by Kent Beck and Erich Gamma. The phrase 'Test-Driven Development' is frequently used to indicate the strong use of unit testing during development, although some think of it as equivalent to 'Test First' development, in which the tests for code are written prior to writing the code. In Test First Development, the test should initially fail (since nothing has been written) and then pass after the code has been written.

For client side languages, JUnit (for Java), DUnit (for Delphi), NUnit and HarnessIt (for dotNet) all provide Unit Test frameworks. The routines %ut and %ut1, included in this patch, provide the same capabilities for unit testing M code. Initially, the client side tests were console based (i.e., not windows, but just text), and that is what %ut provides. For those who like pretty windows, there is an optional GUI front end, MUnit_OSEHRA.exe, available for use.

## 3.1. Getting Started

If you are going to modify sections of your code, or refactor[1], it is best to create a unit test for those areas with which you want to work. Then the unit tests can be run as changes are made to insure that nothing unexpected has changed. For modifications, the unit tests are then written to reflect the new expected behavior and used to insure that it is what is expected. One of the major benefits of unit testing is finding those changes in other parts of your code due to the changes that the modified code made.

The following is a very simple sample routine that covers everything necessary for generating a basic unit test and includes examples of the various calls available:

```
XXXX    ;jli/fo-oak - demo code for a unit test routine ;9/25/03 15:44
        ;;
        ; makes it easy to run tests simply by running this routine and
        ; insures that %ut will be run only where it is present
        I $T(EN^%ut)'="" D EN^%ut("XXXX")
        Q
        ;
STARTUP         ; optional entry point
        ; if present executed before any other entry point any variables
        ; or other work that needs to be done for any or all tests in the
        ; routine. This is run only once at the beginning of processing
        Q
        ;
SHUTDOWN    ; optional entry point
        ; if present executed after all other processing is complete to remove
        ; any variables, or undo work done in STARTUP.
        Q
        ;
SETUP ; optional entry point
        ; if present it will be executed before each test entry to set up
        ; variables, etc.
        Q
        ;
TEARDOWN    ; optional entry point
        ; if present it will be executed after each test entry to clean up
        ; variables, etc.
        Q
        ;
ENTRY1          ; Example for use of CHKEQ call to check two values
        ;
        ; code to generate a test, e.g. to check the sum of 1 and 1
        S X=1,Y=1
```

[1] Clean up the code without changing its behavior, frequently done prior to changing the behavior – see Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Westford, MA: Addison Wesley Longman, Inc.

```
        D CHKEQ^%ut(2,X+Y,"1+1 didn't yield 2") ;
        ;
        ; usage of CHKEQ^%ut
        ;   first argument is the expected value
        ;   second argument is the actual value
        ;   third argument is text to be displayed if the first argument
        ;     and second argument are not equal.
        ;
        ;   Multiple calls to CHKEQ^%ut may be made within one entry
        ;   point. Each of these is counted as a test.
        ;
        ;   Output for a failure shows the expected and actual values
        Q
        ;
ENTRY2          ; Use of CHKTF call to check value for True or False
        ;
        S ERRMSG="Current user is not an active user on this system"
        D CHKTF^%ut($$ACTIVE^XUSER(DUZ)>0,ERRMSG)
        ;
        ; usage of CHKTF^%ut
        ;   first argument is an expression evaluating to true or false value,
        ;   second argument is text to be displayed if the first argument
        ;     evaluates to false.
        ;
        ;   Multiple calls to CHKTF^%ut may be made within one entry
        ;   point. Each of these is counted as a test.
        Q
        ;
ENTRY3          ; Use of CHKTF call to check values that should NOT be equal
        ;
        ; if you want to test something that should fail, use a NOT
        S X=1,Y=3
        D CHKTF^%ut(X'=Y,"indicated 1 and 3 are equal")
        Q
        ;
ENTRY4          ; @TEST - Use of the FAIL call to generate a failure message
        ;
        S X=1+2 I X'=3 D FAIL^%ut("System is doing bad addition on 1+2") Q
        ;
        ; usage of FAIL^%ut
        ;   the argument is text indicating why the failure was identified
        Q
        ;
        ; Other routine names to be included in testing are listed one per line
        ; with the name as the third semi-colon piece on the line and an
        ; optional description of what the routine tests as the fourth semi-
        ; colon piece, if desired this permits a suite of test routines to be
        ; run by simply starting one of the routine the names may be repeated
```

```
        ; in multiple routines, but will only be included once. The first line
        ; without a third piece terminates the search for routine names (which
        ; is why this is above the XTROU tag).
XTROU ;
        ;;XXXY;description of what the routine tests
        ;;XXXZ;
        ;;XXXA
        ;
        ; Entry points for tests are specified as the third semi-colon piece,
        ; a description of what it tests is optional as the fourth semi-colon
        ; piece on a line. The first line without a third piece terminates the
        ; search for TAGs to be used as entry points
XTENT  ;
        ;;ENTRY1;tests addition of 1 and 1
        ;;ENTRY2;checks active user status
        ;;ENTRY3;
        Q
```

Running XXXX as written above results in the following:

```
>D ^XXXX
Referenced routine XXXY not found.
Referenced routine XXXZ not found.
Referenced routine XXXA not found.
…

Ran 1 Routine, 4 Entry Tags
Checked 3 tests, with 0 failures and encountered 0 errors.
>
```

You will not normally see routines that aren't there referenced, since you would not include them. By default, passed tests are shown only with a dot and the results are summarized at the bottom.

To illustrate a failure, change the code on line ENTRY+3 from (X'=Y) to (X=Y). Running XXXX shows that the test now fails. The location of the tag and the comment for failure are shown in the order of the tests:

```
>D XXXX
Referenced routine XXXY not found.
Referenced routine XXXZ not found.
Referenced routine XXXA not found.
..
ENTRY3^XXXX – indicated 1 and 3 are equal


Ran 1 Routine, 4 Entry Tags
```

```
Checked 3 tests, with 1 failure and encountered 0 errors.
>
```

Now change the code on line ENTRY1+3 so that S X=1,Y=1 becomes X=1,Y=1 (removing S<space>. Running XXXX again identifies the error generated due to our typing, as well as continuing on to show the failure we introduced at ENTRY3. The test at ENTRY2 still runs without a problem, as indicated by the lone dot.

```
>D XXXX
Referenced routine XXXY not found.
Referenced routine XXXZ not found.
Referenced routine XXXA not found.

ENTRY1^XXXX - tests addition of 1 and 1 - Error: ENTRY1+3^XXXX:1, %DSM-E-
COMAND,
 bad command detected
.
ENTRY3^XXXX - indicated 1 and 3 are equal


Ran 1 Routine, 4 Entry Tags
Checked 3 tests, with 1 failure and encountered 1 error.
>
```

If the code at ENTRY4+2 is now modified to S X=1+1 and running it causes the FAIL call to be used.

```
>D XXXX
Referenced routine XXXY not found.
Referenced routine XXXZ not found.
Referenced routine XXXA not found.

ENTRY1^XXXX - tests addition of 1 and 1 - Error: ENTRY1+3^XXXX:1, %DSM-E-
COMAND,
 bad command detected
.
ENTRY3^XXXX - indicated 1 and 3 are equal

ENTRY4^XXXX - example of FAIL^%ut call - System is doing bad addition on 1+2


Ran 1 Routine, 4 Entry Tags
Checked 4 tests, with 2 failures and encountered 1 error.
>
```

Restoring S<space> on line ENTRY1+3, and changing X=1 to X=2 and running it shows the output of the CHKEQ call.

```
>d XXXX
Referenced routine XXXY not found.
Referenced routine XXXZ not found.
Referenced routine XXXA not found.

ENTRY1^XXXX - tests addition of 1 and 1 - <2> vs <3> - 1+1 didn't yield 2
.
ENTRY3^XXXX - indicated 1 and 3 are equal

ENTRY4^XXXX - example of FAIL^%ut call - System is doing bad addition on 1+2


Ran 1 Routine, 4 Entry Tags
Checked 4 tests, with 3 failures and encountered 0 errors.
>
```

That covers the basics of generating a unit test routine to use with %ut. For sections of code performing calculations, etc., this is all that will be required. For other cases, depending upon database interactions or of input and output via something like the RPCBroker, other approaches to creating usable tests are required. These 'objects,' which can be used for consistency in such units tests, are generally referred to as 'Mock Objects.'

## 3.2. M–Unit Test Dos and Don'ts

You do not want to include any code which requires user input. You want the tests to be able to run completely without any user intervention other than starting them. By referencing other, related unit test routines within the one that is started, you can build suites of tests that can be used to cover the full range of your code.

## 3.3. M–Unit Test Definitions

Supported References in %ut are EN, CHKTF, CHKEQ, FAIL, CHKLEAKS, ISUTEST and RUNSET.

The entry point EN^%ut(ROUNAME,VERBOSE,BREAK) starts the unit testing process. The first argument is required and provides the name of the routine where the testing should be started. That routine must have at least one test entry point (and possibly more) either specified in the line(s) immediately following the tag XTENT as the third semi-colon piece on the line OR it can have tags with @TEST as the first text of the comment for the tag line. The second optional argument, VERBOSE, if it evaluates to true (e.g., 1) will turn on verbose mode, which lists each individual test being run as well as its result. The third optional argument, BREAK, if it evaluates to true, will cause the M-Unit Test process to terminate upon a failure or error instead of continuing until all tests have been evaluated before finishing, as it normally does.

A test is performed on a conditional value by calling the entry point CHKTF^%ut(testval,messag) with the first argument the conditional test value (true or false) and the second argument a message that should be displayed indicating what failed in the test.

A test checking two values for equivalence is performed by using the entry point CHKEQ^%ut(expected,actual,messag) with the first argument the expected value, the second argument the actual value, and the third argument the message for display on failure.

The entry point FAIL^%ut(messag) is used to simply generate a failure with the argument as the message to be displayed for the failure. This is normally used in a section of code that would not be expected to be processed.

The entry point DO CHKLEAKS^%ut(CODE,TESTLOC,.NAMEINPT) can be used within unit tests or in a stand alone test for variable leaks (those created within called code that are allowed to leak into the calling environment, unintentionally). The CODE argument would contain a command to be executed in the test for leaks (e.g., "S X=$$NOW^XLFTD()"). The TESTLOC argument would indicate the location under test (e.g., "$$NOW^XLFDT() leak test" or simply "$$NOW^XLFDT"). The NAMEINPT variable is passed by reference, and is an array which contains a list of all variables that the user is passing in and/or expects to be present when the code is finished (the variable X would be in the latter category, since it would then be present). The input is in the form of an array NAMEINPT("VARNAME")="VARVALUE", where VARNAME is the name of a variable, and VARVALUE is the value that is to be assigned to the variable before the contents of CODE is to be executed. When run in a unit test environment, variables that are present after the contents of CODE is executed that were not included in NAMEINPT as variables, will be listed as failures. When called outside of a unit test environment, any leaked variables will be listed on the current device.

The entry point ISUTEST^%ut is an extrinsic function used as S X=$$ISUTEST^%ut, and if a unit test is currently running, it will return a value of 1, otherwise it returns a value of zero. This can be used to select code to be run based on whether it is currently being tested (or something else that calls it is being tested).

For those who have problems keeping track of routine names for unit testing and which application they are associated with, the file M-UNIT TEST GROUP (#17.9001) can be used to maintain groups of unit test routines with the edit option "utMUNIT GROUP EDIT" (M-Unit Test Group Edit). These may be run from the option "utMUNIT GROUP RUN" (Run M-Unit Tests from Test Groups), or from a Supported Reference [D RUNSET^%ut(setname)], or from the GUI client described below (click the 'Select Group' button).

While the order of processing within M unit tests may actually be fairly constant, or at least appear to be so, it is preferable to have the unit tests independent of the order in which they are run. Having dependencies between tests can result in problems if the order were to change or if changes are made in the test being depended upon. While STARTUP and SETUP tags are available, there are those who recommend caution even in using them[2].

# 4. MUnit.exe

The GUI MUnit application provides a visually interactive, rapid method for running unit tests on M code. The GUI interface for M UNIT is available as a zip file (MUnit_OSEHRA.zip). It should be saved and the file unzipped into any desired directory. If desired, a shortcut containing specifications for a server and port (e.g, munit.exe s=server.myaddress.com p=9200) can be set up to start MUnit.exe.

- Start the application either double clicking on it or the shortcut.

---

[2] Osherove, R. (2014). *The Art of Unit Testing with Examples in C#, Second Edition.* Shelter Island, NY: Manning Publications Co., p. 34-35.

- Select or Change the server/port specifications if necessary, and click on the 'Connect' button.
- After specifying the server address and port, the user can sign on or click the Select Group button to select a unit test from the M-UNIT TEST GROUP file (#17.9001) as shown here (Figure 1).

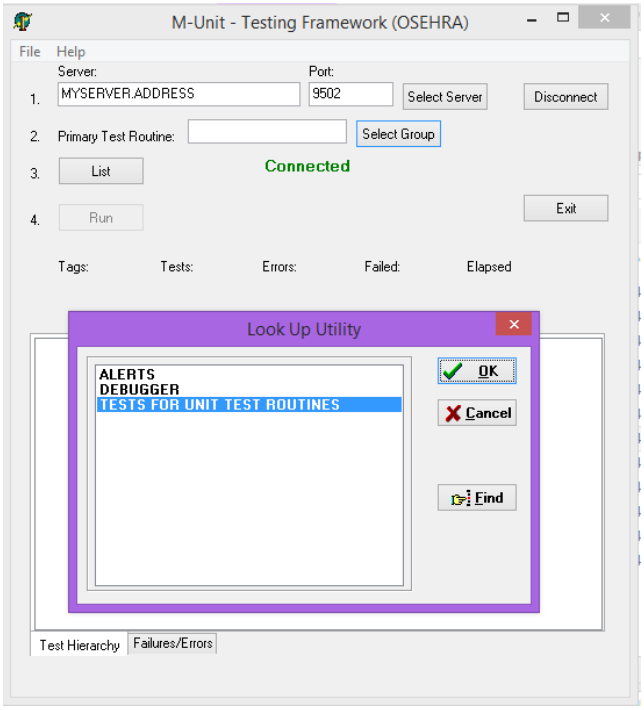


**Figure 1. Selection of an M-Unit test**

You could also simply enter the name of a unit test routine in the Primary Test Routine field and click on List. This will bring up a list of the routines and tags in the unit test run (Figure 2).

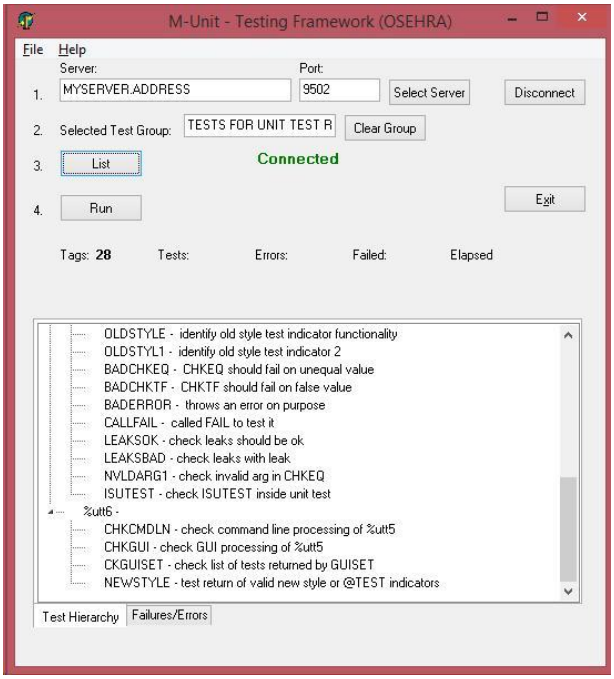


**Figure 2. List of Unit tests selected for running**

Clicking the Run button will run the unit tests, resulting in a bar which is green if all tests pass or red if any failures or errors are encountered (Figure 3).

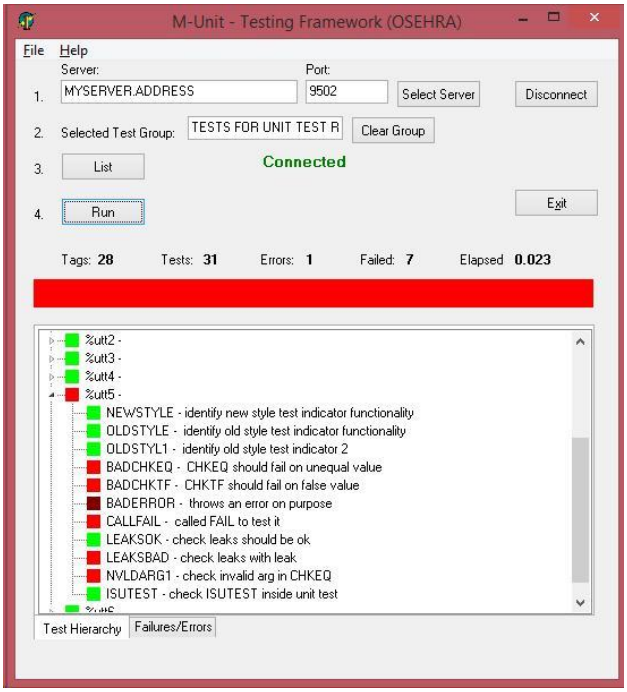


**Figure 3. The unit tests run with failures**

If failures or errors are encountered, clicking on the Failures/Errors tab at the bottom of the listing opens a display of specific information on the problems (Figure 4).

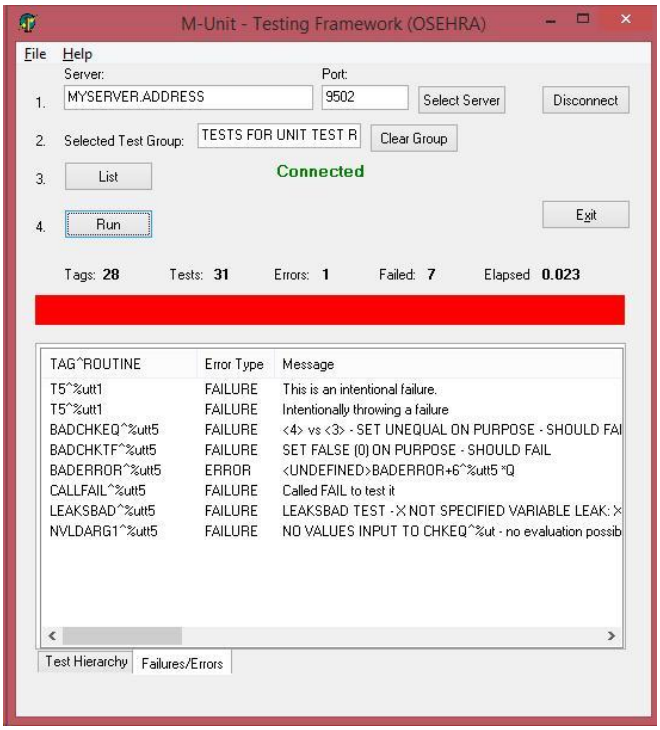


**Figure 4. Specifics on failed tests or errors**

In the case shown (Figure 4), all of the failures are intentional. Usually, failures and/or errors are not intentional and the user can then edit the routine, and save the changes, then simply click on the Run button again to see the effect of the changes.

To select a new unit test, the user would click on the Clear Group button, then again either select another group or as shown in Figure 5, entering the name of a unit test routine (ZZUXQA1 and related routines are not included with the M-Unit Test code and is shown only as an example) and clicking on the List button.

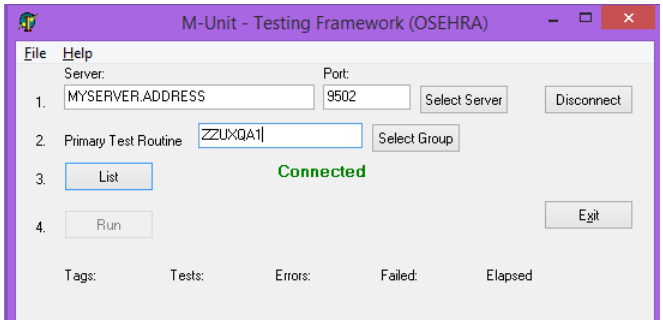


**Figure 5.Specification of unit tests by routine name**

Again, clicking the Run button will run the unit tests (Figure 6). This figure shows the desired result, a green bar meaning that all tests passed.

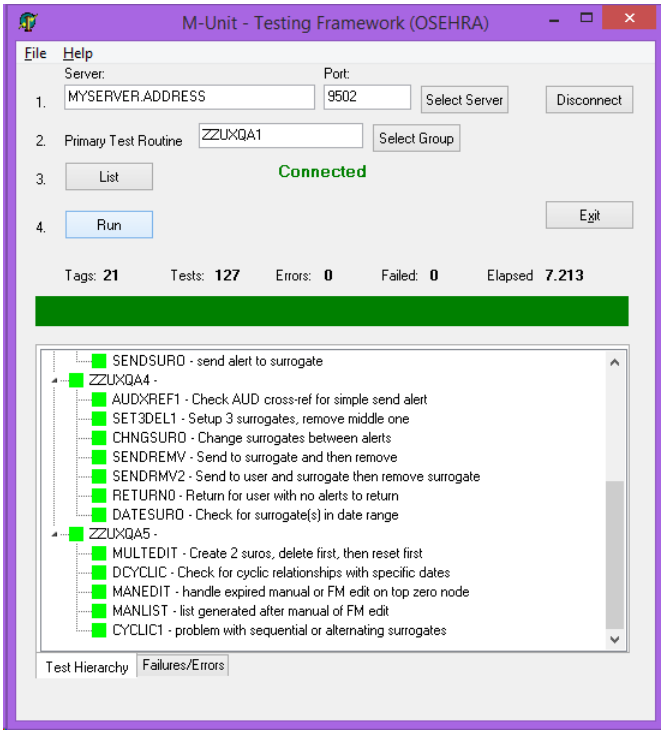


**Figure 6. Result from the second group of unit tests**

# 5. Installation of the M–Unit Software

The installation software for the M–Unit Tools is usually available as either a PackMan message or as a KIDS build file. The basic M–Unit Tools could be loaded from routines only if the usage will be at the command line only.

For installation from a PackMan message:

- open the message and, at the prompt to 'Enter message action', enter X for Xtract KIDS,
- it will then prompt to 'Select PackMan Action', enter 6 for 'INSTALL/CHECK MESSAGE' and follow the subsequent prompts.

For installation from a KIDS build file:

- from the EVE ('System Manager Menu') menu, select:
  - 'Programmer Options'
  - the KIDS ('Kernel Installation & Distribution System') menu
  - and 'Installation', followed by 1 or 'Load a Distribution,'
- at the prompt, enter the host file name (and if using Cache, the directories, if not in the current namespace directory),
- then enter 6 (for 'INSTALL/CHECK MESSAGE') and follow the subsequent prompts.
- The following is an example installation on a Cache system.

```
Select Kernel Installation & Distribution System Option: INStallation

Select Installation Option: 1  Load a Distribution
Enter a Host File: MASH-0_1-0.KID

KIDS Distribution saved on Aug 05, 2014@19:45:42
Comment: M-Unit Functionality

This Distribution contains Transport Globals for the following Package(s):
Build MASH*0.1*0 has been loaded before, here is when:
     MASH*0.1*0   Install Completed
                was loaded on Aug 04, 2014@12:54:12
OK to continue with Load? NO// y  YES

Distribution OK!

Want to Continue with Load? YES// y  YES
Loading Distribution...

  MASH*0.1*0
Use INSTALL NAME: MASH*0.1*0 to install this Distribution.

 Select Installation Option: install Package(s)
Select INSTALL NAME: MASH*0.1*0       Loaded from Distribution     Loaded from D
istribution  8/5/14@20:00:59
    => M-Unit Functionality  ;Created on Aug 05, 2014@19:45:42

This Distribution was loaded on Aug 05, 2014@20:00:59 with header of
  M-Unit Functionality  ;Created on Aug 05, 2014@19:45:42
  It consisted of the following Install(s):
    MASH*0.1*0
Checking Install for Package MASH*0.1*0

Install Questions for MASH*0.1*0

Incoming Files:

  17.9001   M-UNIT TEST GROUP  (including data)

Want KIDS to Rebuild Menu Trees Upon Completion of Install? YES// NO

Want KIDS to INHIBIT LOGONs during the install? YES// NO
Want to DISABLE Scheduled Options, Menu Options, and Protocols? YES// NO

Routine:      ut1 Loaded, Saved as     %ut1
Routine:     utt1 Loaded, Saved as    %utt1
Routine:     utt2 Loaded, Saved as    %utt2
Routine:     utt3 Loaded, Saved as    %utt3
```

```
Routine:     utt4 Loaded, Saved as    %utt4
Routine:     utt5 Loaded, Saved as    %utt5
Routine:     utt6 Loaded, Saved as    %utt6

 Updating Routine file...

 Updating KIDS files...

 MASH*0.1*0 Installed.

 NO Install Message sent
------------------------------------------------------------------------------
              ----------------------------------------
  100%       x           25           50           75           x
Complete     ----------------------------------------


Install Completed

Select Installation Option:
```

- The following is an example of part of an installation on a Linux system without any problems. The initial part is basically the same as above, and the listing is shown from where it changes just before it starts converting routines to %ut*.

```
In the next section, as it tries to copy the ut* routines
to %ut* routines watch for text that indicates the following:

cp: cannot create regular file `/_ut.m': Permission denied

If this is seen, respond Yes at the prompt after the attempt:
   Press ENTER to continue:

Routine:       ut Loaded, Saved as      %ut

Routine:      ut1 Loaded, Saved as     %ut1

Routine:     utt1 Loaded, Saved as    %utt1

Routine:     utt2 Loaded, Saved as    %utt2

Routine:     utt3 Loaded, Saved as    %utt3

Routine:     utt4 Loaded, Saved as    %utt4
```

```
Routine:     utt5 Loaded, Saved as    %utt5

Routine:     utt6 Loaded, Saved as    %utt6

Routine:  uttcovr Loaded, Saved as %uttcovr

  Your entry on the next line may not echo
If error text was seen enter Y (and RETURN):  NO//
```

- The extra text is present since there is at least one Linux version that has a problem with converting the ut* names the files are transported as to the %ut* names that are used on the system. If the listing of the name conversion looks like the following, then responding y or Y followed by return to the prompt will list the following information for manually converting the files.

```
In the next section, as it tries to copy the ut* routines
to %ut* routines watch for text that indicates the following:

cp: cannot create regular file `/_ut.m': Permission denied

If this is seen, respond Yes at the prompt after the attempt:
   Press ENTER to continue:
 cp: cannot create regular file `/_ut.m': Permission denied

Routine:       ut Loaded, Saved as      %ut
 cp: cannot create regular file `/_ut1.m': Permission denied

Routine:      ut1 Loaded, Saved as     %ut1
 cp: cannot create regular file `/_utt1.m': Permission denied

Routine:     utt1 Loaded, Saved as    %utt1
 cp: cannot create regular file `/_utt2.m': Permission denied

Routine:     utt2 Loaded, Saved as    %utt2
 cp: cannot create regular file `/_utt3.m': Permission denied

Routine:     utt3 Loaded, Saved as    %utt3
 cp: cannot create regular file `/_utt4.m': Permission denied

Routine:     utt4 Loaded, Saved as    %utt4
 cp: cannot create regular file `/_utt5.m': Permission denied

Routine:     utt5 Loaded, Saved as    %utt5
```

```
 cp: cannot create regular file `/_utt6.m': Permission denied

Routine:     utt6 Loaded, Saved as     %utt6
 cp: cannot create regular file `/_uttcovr.m': Permission denied

Routine:  uttcovr Loaded, Saved as %uttcovr


  Your entry on the next line may not echo
If error text was seen enter Y (and RETURN):  NO//

*** An error occurred during renaming of routines to %ut*.
*** The renaming has been seen to fail on one type of Linux system.
*** In this case, at the Linux command line copy each ut* routine
*** (ut, ut1, utt1, utt2, utt3, utt4, utt5, utt6, and uttcovr)
*** to _ut* (e.g., 'cp ut _ut', 'cp ut1 _ut1', etc. to 'cp uttcovr _uttcovr').
*** Then in GT.M use the command  'ZLINK %ut', then 'ZLINK %ut1', etc., these
*** may indicate an undefined local variable error, but continue doing it.
*** When complete, use the M command 'DO ^%utt1' to run the unit tests on
*** the %ut and %ut1 routines to confirm they are working

  Press Enter to continue:
```

# 6. Running M-Unit Tests

Once the installation is complete, you can verify that the %ut test framework has been installed correctly by running the supplied test routines.

There are several routines that can be run to test various areas of the M-Unit functionality. Running the routine %utt1 from the top (i.e, DO ^%utt1) will run the verbose mode of the basic tests. The routine %utt6 when run from the top (DO ^%utt6) or in verbose mode (DO VERBOSE^%utt6) runs a variety of tests for APIs covering the command line, options, and GUI, and the routine %uttcovr is run from the top to perform coverage analysis (only available on GT.M) for the %ut and %ut1 routines. Entering the following command at the VistA command prompt will run the basic tests (and the 7 failures and 1 error are caused on purpose to test specific parts of the code associated with failures and errors):

```
>D EN^%ut("%utt1")
....
T5^%utt1 - Error count check - This is an intentional failure.
.
T5^%utt1 - Error count check - Intentionally throwing a failure
.............
BADCHKEQ^%utt5 -  CHKEQ should fail on unequal value - <4> vs <3> - SET
UNEQUAL
ON PURPOSE - SHOULD FAIL
```

```
BADCHKTF^%utt5 -  CHKTF should fail on false value - SET FALSE (0) ON
PURPOSE -
SHOULD FAIL

BADERROR^%utt5 -  throws an error on purpose - Error:
150372778,BADERROR+6^%utt5
,%GTM-E-EXPR, Expression expected but not found

CALLFAIL^%utt5 -  called FAIL to test it - Called FAIL to test it

LEAKSBAD^%utt5 - check leaks with leak - LEAKSBAD TEST - X NOT SPECIFIED
VARIABL
E LEAK: X

NVLDARG1^%utt5 - check invalid arg in CHKEQ - NO VALUES INPUT TO
CHKEQ^%ut - no
evaluation possible
....

Ran 5 Routines, 26 Entry Tags
Checked 29 tests, with 7 failures and encountered 1 error.
```

The supplied tests can be run manually, but are also part of the OSEHRA VistA Automated Testing harness. For instructions on how to acquire and run the tests via the OSEHRA harness see the online documentation at:

- https://github.com/OSEHRA/VistA/blob/master/Documentation/ObtainingTestingCode.rst
- and https://github.com/OSEHRA/VistA/blob/master/Documentation/RunningandUploadingTests.rst

Then execute the following CTest command to run the tests:

```
ctest -R UNITTEST_Mash_Utilities
```