mvTest
User Guide

Version 1.3

Brian Leach Consulting Limited

http://www.brianleach.co.uk

Contents

Introduction

mvTest is a suite of programs that provides developers with the tools to create and run a wide variety of automated tests against MultiValue applications.

mvTest provides a framework that can be used for:

- Unit testing
- Regression testing
- Continuous Integration testing
- Volume testing

mvTest goes beyond the traditional testing methods by offering features specific to MultiValue platforms and an approach that fits well with Multivalue development and developers. By embracing the testing philosophy in mvTest, whether as an adjunct to your existing testing regimes or as a means of moving towards a modern test driven development cycle, mvTest can help you to create more robust and scalable applications.

From the Author

As a solo developer and consultant I have become ever more acutely aware of the need to use automated testing as part of my product development, especially as my product range has expanded over the years. Like many experienced developers I long resisted the move to automated testing, but since embracing this and retrofitting automated testing into my code I have already seen the up-front cost in time and effort repay itself many times over in the savings in support time and greater confidence in my products before they are released.

When I embarked on this journey, however, I could not find a tool set that worked well with the back end of my applications whilst giving me the flexibility I need. Regular scripting tools could help automate some activities but could not easily check the results; unit testing tools were too restrictive and mostly only worked with client side code; external testing frameworks forced a heavy learning curve and did not sit well with multivalued applications.

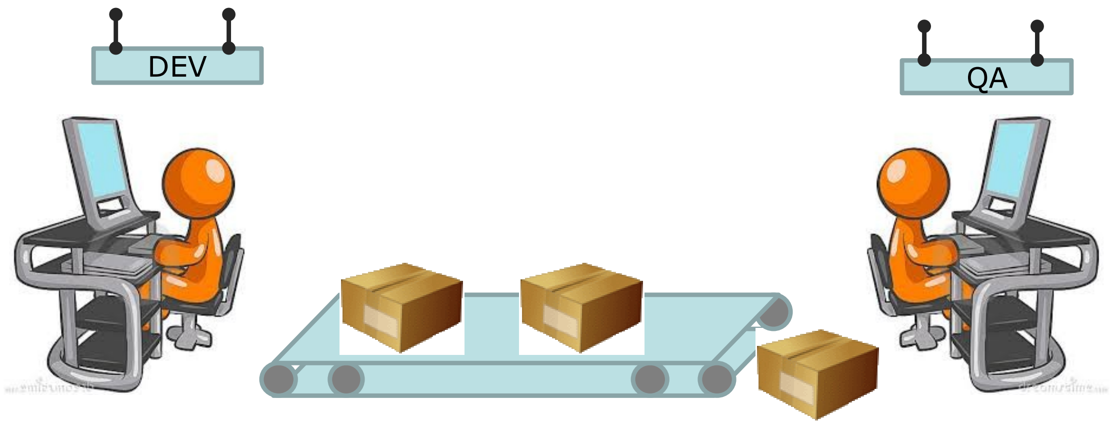
mvTest was my response, and I look forward to hearing your feedback to further progress the product.

Testing

Good testing depends on good design.

In order to test a system, it must be testable – that is, the developers must have thought about how it should be tested and must be involved in the testing from the outset.

The old days in which developers wrote code and threw it over the wall to QA to break are long gone. Agile practices, eXtreme programming any sheer sensibleness have seen to it that developers are now at the forefront of the testing regime.



The old days when MultiValue developers worked in their own little silos and lobbed packages over the wall to QA and UAT are long gone! Developers are expected to unit test their code, and for good reasons.

- Testing is the best way to understand your code.
- Testing documents your code.
- Testing gives you the confidence to refactor your code.

Frankly, testing leads to better code.

Test First Approaches

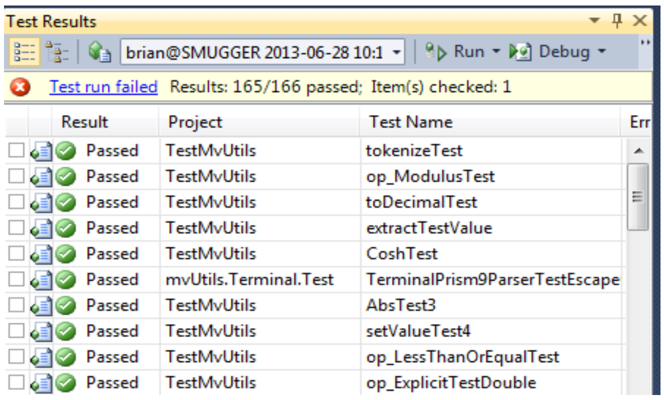
The extreme of this is the Test First or TDD (Test Driven Development) approach that is beloved of many Agile coaches (though it's worth stating that TDD did not begin with Agile). Test Driven Development puts the testing first, encouraging developers to think about and to design and code their tests before they start work on their code.

The rationale behind this is simple – if you build your tests up front it forces you to think about you code, how testable it will be, and how well you have understood the requirements. This also goes along with another often-quoted rationale: 'this far, and no further'. By creating the code to pass the tests, and not the other way round, you can ensure that your work is directly confronting the job in hand and not engaging in flights of developmental fancy at your employers' expense.

You do not need to adopt a Test First approach to use mvTest, but it helps.

Automated Testing

One of the most successful movements in modern programming has been an increased focus for all shapes and sizes of projects on the use of automated testing. Whilst automated testing has always been possible, the creation of solid and easy-to-use open source testing tools such as the various xUnit (jUnit, dUnit, NUnit and so forth) testing platforms, and the inclusion of ever more sophisticated unit testing tools in development environments such as Visual Studio, has opened up this aspect of testing to a very wide range of developers.



Test run failed Results: 165/166 passed; Item(s) checked: 1

Result	Project	Test Name	Err
Passed	TestMvUtils	tokenizeTest	
Passed	TestMvUtils	op_ModulusTest	
Passed	TestMvUtils	toDecimalTest	
Passed	TestMvUtils	extractTestValue	
Passed	TestMvUtils	CoshTest	
Passed	mvUtils.Terminal.Test	TerminalPrism9ParserTestEscape	
Passed	TestMvUtils	AbsTest3	
Passed	TestMvUtils	setValueTest4	
Passed	TestMvUtils	op_LessThanOrEqualTest	
Passed	TestMvUtils	op_ExplicitTestDouble	

MS Test – catching a (deliberately) failing test inside the mvTest client.. the author practicing what he preaches.

For traditional programmers brought up without the benefits of these frameworks, creating automated tests can at first appear to be an overhead and a block on getting on with the work. If you are in that camp, just ask yourself how often you have had to go back and change any of your code after it has been released – whether because of errors, misunderstandings or later changes in functionality. If each release requires consistent testing, the cost of NOT having automated tests in place (both in terms of time and resources, and in the sanity of the developer) can quickly become unsustainable.

Unfortunately, whilst testing frameworks for client side code – java, .NET, Delphi or web based – are easy to come by, the same is not true of testing MultiValue applications or back end code. The best modern MultiValue applications should continue to hold their business logic close to the server, where it should be easily maintained and tested independent of the latest fads in front end presentation. Traditionally, this was done – if at all – by writing programs to test each subroutine call or through long hours of practice entering data (I spent so long over the years typing in test trades for a Metals trading system that I can enter the most complex multi-leg trades without even looking at the screen).

mvTest has been designed to bring that ease of use and functionality to testing MultiValue applications.

Refactoring Code

So does good testing lead to better code? It sure can help!

First, your code needs to be testable. There is no magic bullet that can turn your spaghetti-like steam-driven legacy code into beautifully crafted modern pieces of loveliness, but for new code adopting a unit test strategy forces you into good practices.

To test code, it must be accessible. To unit test code, it should also be broken down into small testable pieces (the Single Responsibility Principle). That does not mean making every line of code into a separate Basic subroutine, but does mean that you should be following the structured programming practices we all learned back in the eighties (or before).

Here are some basic rules worth following, and like all rules they are designed to be broken:

- Separate the business logic from the UI and other dependencies. Yes, it's sometimes tough, but it is extremely liberating when you do.

- If code is generic, or can be reused, pull it into an external subroutine. Then you can unit test that and be assured it will still work when called from anywhere else.

- If code is local, split it into internal subroutines (GoSub) inside a larger subroutine and have a calling mechanism that lets you address the sections individually[1].

- Use INCLUDEs to cover especially tricky pieces of code that you can't separate out.

That's all well and good, but what if you are faced with some 1980s filth that you need to change? That is where refactoring comes in.

Refactoring is the art of changing code without changing what it does. There are many books on the subject, and almost all of them assume you are using Java or .NET. But in many ways the principles are the same: leave the code better than when you found it, and cover it in unit tests before you begin so you can refactor with confidence.

The biggest reason why old, nasty code exists is because developers are too scared to change it. The more you can cover the old in tests, the lower the risk that you will break it when you refactor.

[1] A Case statement at the top of your code is a good way to approach this:

```
SUBROUTINE MySub( Action, InData, OutData, ErrText )

Begin Case
  Case Action = ACTION.DOSTUFF
     GoSub DoStuff
 Case Action = ACTION.DOOTHERSTUFF
   GoSub DoOtherStuff
End Case
```

In this way you can easily drive your subroutine in the combinations needed to test the individual pieces without ending up with so many subroutines that you can never find the one you want.

What should automated testing provide?

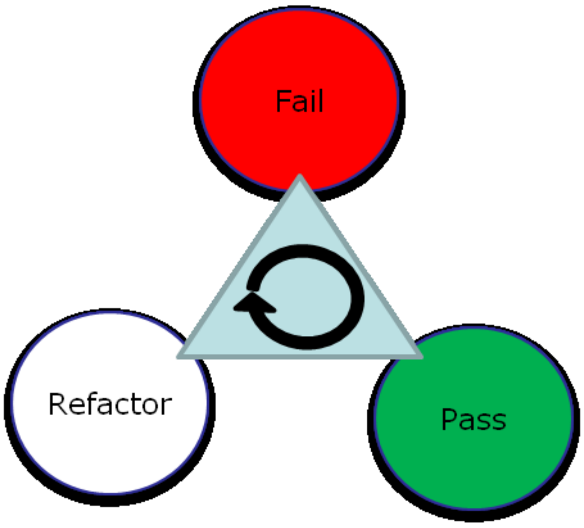
At the simplest level, automated testing such as unit testing provides a repeatable experience for checking the functionality in your system. Each area of functionality should be expressed through its own unit test: in this way, when the functionality changes only the one test need be updated. As the functionality is developed, so the test can be run and re-run proving that the developer is on track until the functionality is complete. It can also provide an at-a-glance view of progress as unit tests are covered and turn from red to green. This is the heart of unit testing and of Test Driven Development.

When new features are added or new releases prepared the whole series of tests can be played to ensure that there are no unintended side-effects. In a perfect world this will test the whole system without the need for intervention: in an imperfect world it will indicate at least an area of coverage that will require no further intervention. The latter is important when retrofitting unit tests into an existing legacy application. This is the heart of regression testing.

Some applications run through a complex build process to prepare them for release. This is not often the case for multivalue applications where everything is broadly held in the same small group of languages, but there may be integration issues with customer-specific routines or software configurations that require parallel testing. These can be handled using continuous integration testing, where the test series are replayed constantly against an integration server.

And finally, whilst automated testing typically focuses on the need to ensure functionality, the same tests can also be harnessed for volume and stress testing. By creating tests that specifically mirror user functionality (complete with lookups, data entry errors and natural pauses) you can create simulations of normal business activity and scale these up across multiple connections to see how your application will behave under different loads and in real life situations.

mvTest has been designed to address all of these situations.



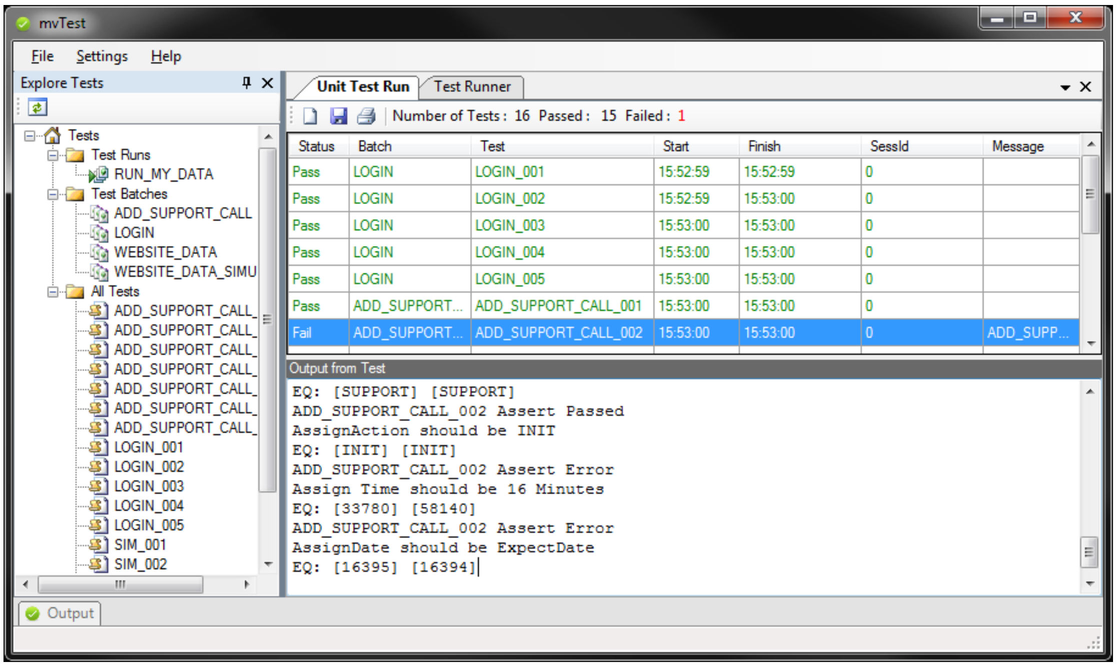
How mvTest Works

mvTest is presented as a Windows based client application that is used to define and organize individual tests, test batches and test runs that can be targeted against a MultiValue database. Additional applications provide command-line driven test runners, a SLIM runner for interfacing with FitNesse for acceptance testing, a UI recorder and a server side test runner.

The tests themselves are written as scripts using a specialized scripting language that offers a level of compatibility with UniVerse Basic[2], but that has also been extended with a number of features that make it easier to run tests, handle assertions, generate or randomize test data and check the results of your scripted activities.

These scripts can be run singly or as batches to test a specific area of a system or to set up the necessary test data before running the unit tests. As well as offering a convenient way to run a series of tests, a batch may be preceded by initialization commands and succeeded by tear down commands: an example might be to call a subroutine to initialize a common block or to clear down a work-file on completion.

Batches in their turn can be organized into test runs. A test run consists of a number of batches that can be run in a prescribed or random order. Test runs add further control: these can be repeated multiple times and at intervals, can be run by multiple sessions in parallel for volume testing, and can also be separately initialized.



Number of Tests : 16 Passed : 15 Failed : 1

Status	Batch	Test	Start	Finish	SessId	Message
Pass	LOGIN	LOGIN_001	15:52:59	15:52:59	0	
Pass	LOGIN	LOGIN_002	15:52:59	15:53:00	0	
Pass	LOGIN	LOGIN_003	15:53:00	15:53:00	0	
Pass	LOGIN	LOGIN_004	15:53:00	15:53:00	0	
Pass	LOGIN	LOGIN_005	15:53:00	15:53:00	0	
Pass	ADD_SUPPORT...	ADD_SUPPORT_CALL_001	15:53:00	15:53:00	0	
Fail	ADD_SUPPORT...	ADD_SUPPORT_CALL_002	15:53:00	15:53:00	0	ADD_SUPP...

Output from Test

```
EQ: [SUPPORT] [SUPPORT]
ADD_SUPPORT_CALL_002 Assert Passed
AssignAction should be INIT
EQ: [INIT] [INIT]
ADD_SUPPORT_CALL_002 Assert Error
Assign Time should be 16 Minutes
EQ: [33780] [58140]
ADD_SUPPORT_CALL_002 Assert Error
AssignDate should be ExpectDate
EQ: [16395] [16394]
```

mvTest Client Test Run results

[2] And therefore, other MultiValue BASIC variants.

Test Script Types

The test scripts used in mvTest generally fall into one of two categories:

The first type of scripts (“direct access scripts”) perform direct database activities such as calling subroutines, running programs and checking the results, and reporting back on the status of the various checks in the shape of assertions. These are typically used for unit testing and regression testing and focus on the underlying functionality. These direct access scripts can also be used to perform back end testing of client/server and web based applications, where the front end testing can be handled by standard xUnit frameworks.

The second type of scripts (“user interface scripts”) includes those that simulate user activity through a traditional text based interface. mvTest includes terminal and TELNET emulation features and commands for scripting and interpreting terminal based activity. These run hand in hand with the direct data access features, so that you can both run your user interface tests and directly interface with the database to check their results. These scripts are typically used for regression and volume testing.

For example, you may wish to unit test an application written in a terminal based 4GL such as SB+. For this you can use the user interface scripts to navigate through the SB+ login and menu system, walk through data entry screens and shell out process calls, just in the same way that a regular user would approach your application. As you do so, you can check what is being presented on the screen, can take different paths depending on the error messages raised by your test data, and report back to the test runner.

At the same time, you may have parts of your application that are designed to be accessed through a web page. Here the direct access scripts are more appropriate: your web pages should be built to call subroutines or data objects, and so the same routines can be called from your test scripts and the results (including JSON formatted data) spot-checked alongside any resulting changes to the database.

Scripting Engines

In order to execute the test scripts, mvTest contains a number of separate script runners: including those built into the mvTest client and the other one server side.

> The ***client side script runner*** opens a client (e.g. UniObjects) connection to perform direct access operations such as reading data or calling subroutines. This is handled separately from the main connection used by mvTest to manage the tests, batches and test runs, so as to counter the effects of subroutine caching in some database types (especially for UniVerse) and to allow for volume testing.
>
> In addition, the client side script runner includes an advanced and fully featured built-in terminal emulator that acts as a conduit for user interface tests. This is only used if the script demands.

> The ***server side script runner*** contains a similar script engine but runs on the server so that it performs any direct accesses within the database itself. If you require user interface testing, it too opens a local socket connection so as to act as a telnet client and emulates a terminal in memory.

> The ***console runner*** embeds the regular client side runner in a simpler console application. This is suitable for launching from automated client side tools such as standard build monitors or test engines. It includes all the features of the client side runner and can output to a number of different formats.

> The ***SLIM runner*** is built to be called from an industry standard acceptance testing framework called FitNesse. This is an open source tool that acts as a collaboration platform for capturing and executing acceptance tests. FitNesse uses a protocol called SLIM to talk to specially-crated external test drivers.

The ***client side script runner*** is the better choice for creating and controlling tests and for running tests when you are present to watch the testing activity. It also offers specific support for volume tests and has some additional scripting functionality that is not available to the server side script runner.

The **console runner** or **server side script runner** is the better choice for unattended running such as CI testing or wide scale regression testing. This is generally quicker as it does not need to report back to the client, and tests can be run and scheduled using phantoms (subject to the restriction that phantoms opening a socket connection will consume a database licence on some platforms). The server runner also allows you to debug the code under test or to see runtime errors that may be swallowed by an UniObjects connection.

Why Scripting?

Effective automated testing is more than a set of simple activities. Effective testing means opening up the potential to run complex tests, including handling real world situations such as responding to locking conditions within volume tests, handling errors from simulated test data or following through a thread of activities all sharing a newly created record key.

Take the case of a support system. You might want to test the tracking of a support call right through from the initial submission where a new ticket number will be raised, the assignment of a support resource and category, a required response time based on rules around the call and the customer, through to a set of actions and responses simulating user feedback until the call is finally closed.

To test this effectively you would need to track the call number and the various business rules that inform each stage. You would also need to check back against the database to make sure the right assignments have been made, emails generated and support diaries updated. And you might want to run this with randomized series of actions or stress test it with a large number of submissions.

To do this requires a level of sophistication that borders on programming.

Scripting

You could do this using regular programming for most cases, but that can be hard work! A scripting language offers the opportunity to extend the language to offer features directly related to testing, such as assertions and special functions for checking data consistency and creating or managing test data.

Examples in the mvTest scripting language include the **ONE()** function that take the heavy lifting out of generating random test data by supplying one random entry from variety of different source specifications (lists of options, file keys, ranges of values) and **Data sources** used to randomly access data from comma delimited or other prepared files.

Similarly the **CHECK()** function combines a variety of structural validation tests including range and pattern matching, file relations, data typing and option lists into a single easy function. The **ASSERT** statement is the base for a whole group of statements that handle test assertions to feed back a pass or fail status on the test.

User Interface Testing

The scripting language also supports the user interface testing through a logical terminal connection with statements for supplying input, waiting for text or capturing areas of the virtual screen.

Some of these can simplify what is quite difficult to achieve behind the scenes: the WaitUntil statement, for example, waits until a given condition is met, such as the cursor appearing on a specific location for a prompt.

Platform Differences

Consider an application that needs to run on UniVerse, UniData and D3. You will already have to handle differences in your application logic, but using a script for the tests allows you to remove some of those differences in your testing.

So, for example, if you want to test that your application has written a comma separated file, you can use the script CLIENT object methods to access the server file system whilst hiding away the differences between ReadSeq, OSBREAD and %read calls.

Management

Finally, scripts can be easily re-used, copied and shared, and can become part of your source management along with the programs they are testing. Scripts are stored server side and so can also be version managed alongside the programs and features that they target.

mvTest even includes an API for interfacing to source control systems directly from the mvTest client.

Whilst scripts form the heart of mvTest, running individual scripts in isolation can only take you so far in your testing environment. The next step is to organize these tests based on workflows to match the operations performed in your system.

mvTest organizes the scripts into batches (ordered tests) that naturally follow a thread of related activities with their own initialization and teardown commands. A batch generally represents one workflow or set of related activities, for example:

- Entering a support call and checking a confirmation email is sent out.
- Entering a support call; checking it is raised correctly; that a minimum response time is applied based on contract terms; that it is assigned to a support person.
- Entering a support call; attaching a file to the call and checking that it is correctly uploaded.
- Entering a support call; sending a message to request further details; accepting the details and closing the call.

To provide even better coverage and to support integration testing, these batches can then organized into test runs that can be launched under a variety of different conditions with different parameter data passed e.g. to test customer-specific variations.

The test run allows additional levels of control to be exercised around the script running at each stage: the need to run routines before or after to set up the conditions for a test, batch, or run; logging and COMO operations; notifications; randomizing batches and volume testing.

Here are the four levels of testing in mvTest:

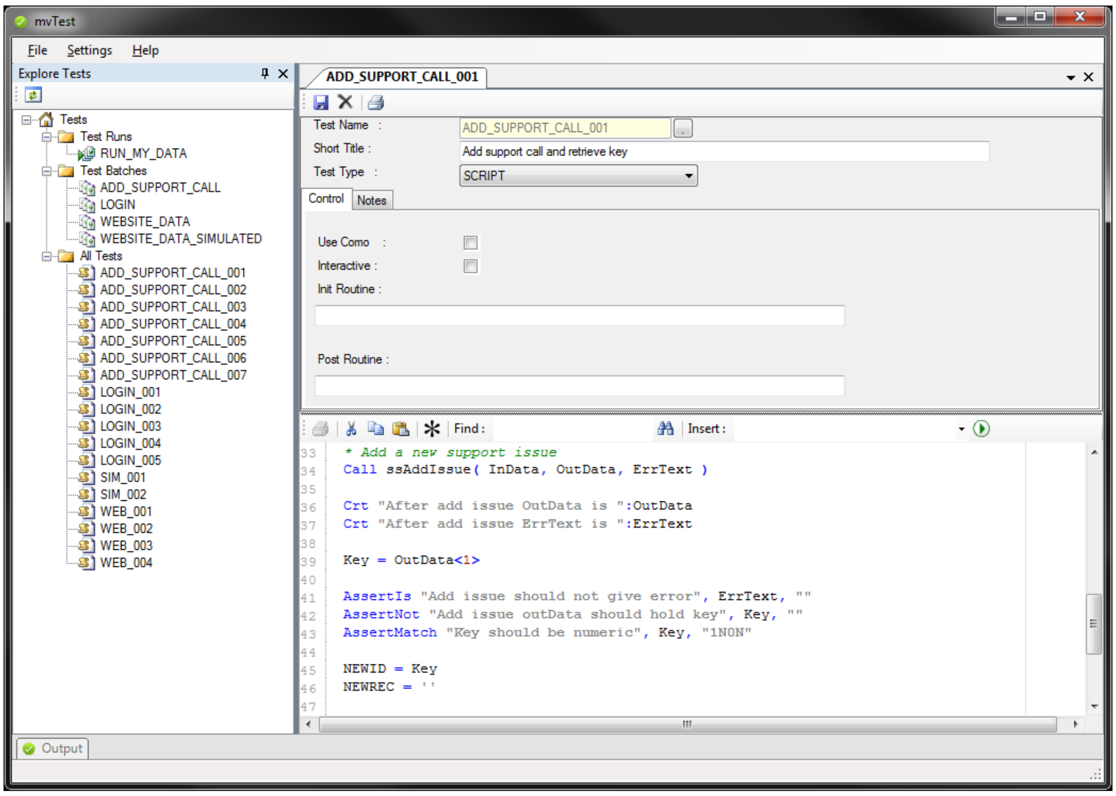
- Unit tests and UI Tests
- Workflows as Test Batches
- Integration flows as Test Runs
- Volume Tests based on Test Runs

Each test consists of control information that dictates how the script should be set up and run and a script of actions to perform written in a dedicated scripting language[3].

Creating a new Test

Tests are stored in the account in which they should run in the TESTS and TEST_SCRIPTS files. You can create test scripts using any standard editor, but it is most convenient to use the mvTest client to build these.

Use the File -> New -> Test option to create a new test:



Test Name : ADD_SUPPORT_CALL_001
Short Title : Add support call and retrieve key
Test Type : SCRIPT

Use Como :
Interactive :
Init Routine :
Post Routine :

```
* Add a new support issue
Call ssAddIssue( InData, OutData, ErrText )

Crt "After add issue OutData is ":OutData
Crt "After add issue ErrText is ":ErrText

Key = OutData<1>

AssertIs "Add issue should not give error", ErrText, ""
AssertNot "Add issue outData should hold key", Key, ""
AssertMatch "Key should be numeric", Key, "1N0N"

NEWID = Key
NEWREC = ''
```

Tests must be compiled before use: this turns the test script into pseudo-code that is stored in the TEST_OUT file. The Compile button will compile the test directly from the client, or you can use the TSC (Test Script Compiler) command at TCL.

Editing an Existing Test

The All Tests folder in the Explorer tree contains a list of all the tests in your account, and additionally each batch entry in the tree can be expanded to show the list of tests currently attached to that batch.

Double click a test in the tree to load it into the Test Editor.

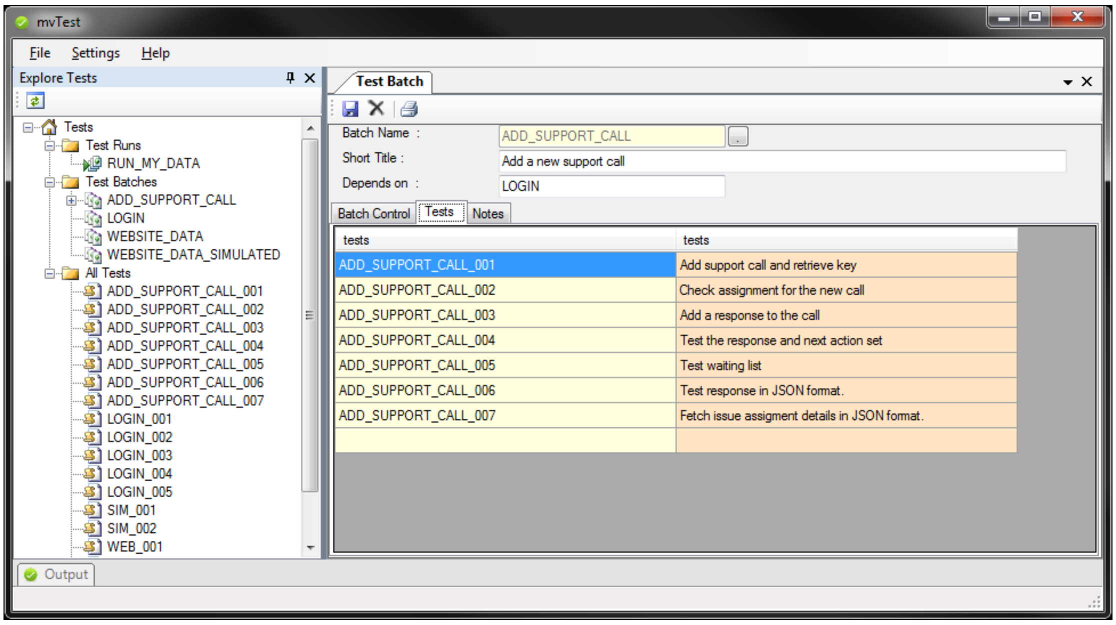
[3] Today all tests in mvTest are handled using scripts: other test types will be added in future.

Test Batches

Whilst individual tests can be run singly, it is more useful to organize your tests into batches.

A batch is a series of one or more tests that naturally belong together and typically form a coherent set: the tests in the batch are always performed in a set order.

Batches also include start-up and shutdown commands, which are useful when you need to set up the testing environment – typically this may be used to initialize common blocks needed by your application.



- Tests
  - Test Runs
    - RUN_MY_DATA
  - Test Batches
    - ADD_SUPPORT_CALL
    - LOGIN
    - WEBSITE_DATA
    - WEBSITE_DATA_SIMULATED
  - All Tests
    - ADD_SUPPORT_CALL_001
    - ADD_SUPPORT_CALL_002
    - ADD_SUPPORT_CALL_003
    - ADD_SUPPORT_CALL_004
    - ADD_SUPPORT_CALL_005
    - ADD_SUPPORT_CALL_006
    - ADD_SUPPORT_CALL_007
    - LOGIN_001
    - LOGIN_002
    - LOGIN_003
    - LOGIN_004
    - LOGIN_005
    - SIM_001
    - SIM_002
    - WEB_001

Batch Name : ADD_SUPPORT_CALL
Short Title : Add a new support call
Depends on : LOGIN

tests	tests
ADD_SUPPORT_CALL_001	Add support call and retrieve key
ADD_SUPPORT_CALL_002	Check assignment for the new call
ADD_SUPPORT_CALL_003	Add a response to the call
ADD_SUPPORT_CALL_004	Test the response and next action set
ADD_SUPPORT_CALL_005	Test waiting list
ADD_SUPPORT_CALL_006	Test response in JSON format.
ADD_SUPPORT_CALL_007	Fetch issue assigment details in JSON format.

Test Batch Definition

To add a new test batch, click the File -> New -> Batch menu item. You can edit an existing batch by double clicking the batch entry in the tree.

Note that each batch in the tree can be expanded to show the tests within the batch. This makes it easier to find a test along a large number of similar tests.

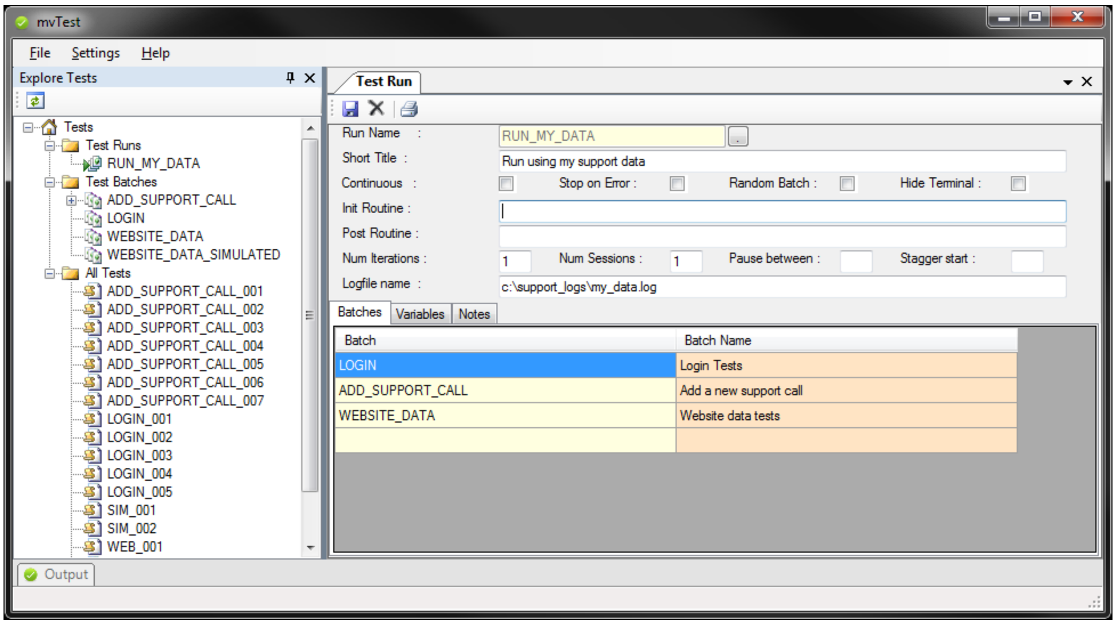
Test batches can be tagged with a user-defined name. These allow similar batches or batches that have a similar pattern of use to be selected for a test run. For example, you might want a test run to test a module of your system (LEDGER) or to separate out those batches that you want to include in your regular CI process from those that are run less often, as part of an overnight or pre-release build.

Batches are made up of one or more tests. These tests are explicitly added to batch, since the order of these may be significant in handling a particular workflow. You can use the F3 lookup to select a test to add, rather than having to explicitly type in the test name.

Test Runs

At the top of the tree, a number of Test Batches may be organized into a Test Run.

A Test Run consists of one or more test batches, which may optionally be randomized in their order before execution. A test run can be set to iterate once through the sequence, a number of times for volume testing or as part of CI or automated build testing. The Pause Between sets a pause in seconds between each run.



- Tests
  - Test Runs
    - RUN_MY_DATA
  - Test Batches
    - ADD_SUPPORT_CALL
    - LOGIN
    - WEBSITE_DATA
    - WEBSITE_DATA_SIMULATED
  - All Tests
    - ADD_SUPPORT_CALL_001
    - ADD_SUPPORT_CALL_002
    - ADD_SUPPORT_CALL_003
    - ADD_SUPPORT_CALL_004
    - ADD_SUPPORT_CALL_005
    - ADD_SUPPORT_CALL_006
    - ADD_SUPPORT_CALL_007
    - LOGIN_001
    - LOGIN_002
    - LOGIN_003
    - LOGIN_004
    - LOGIN_005
    - SIM_001
    - SIM_002
    - WEB_001

Run Name :	RUN_MY_DATA						
Short Title :	Run using my support data						
Continuous :	☐	Stop on Error :	☐	Random Batch :	☐	Hide Terminal :	☐
Init Routine :							
Post Routine :							
Num Iterations :	1	Num Sessions :	1	Pause between :		Stagger start :	
Logfile name :	c:\support_logs\my_data.log						

Batch	Batch Name
LOGIN	Login Tests
ADD_SUPPORT_CALL	Add a new support call
WEBSITE_DATA	Website data tests

Test runs are at the heart of volume testing: a test run can start a number of concurrent sessions, can stagger the start times and can provide additional control over the runtime environment.

You can add batches to a run in one of three ways:

The Select All Batches option does exactly what it says, and will run all the batches in ascending order of the batch name.

The Select Batches with Tag option lets you select a series of batches dynamically that you have tagged with a specific name, for example OVERNIGHT for an overnight CI run. The naming conventions are up to you.

Finally you can add specific batches by name or by using the F3 lookup for the grid below.

Customized Runs

Runs also allow you to customize testing for different environments.

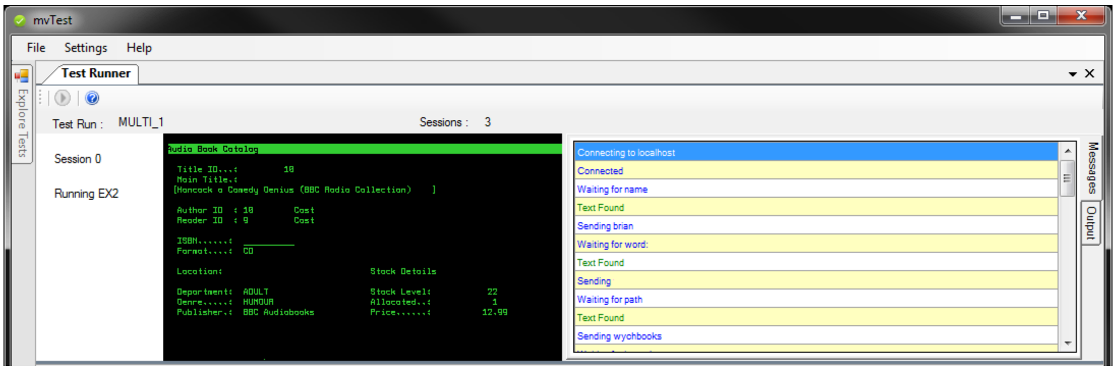
One important feature of Test Runs is the support for localized values that can be used to parameterize tests. The **Run Variables** tab holds a list of keys and values that can be retrieved using the script **GET()** function: this might, for example, be used to customize test data for different client or departmental sites whilst retaining the same batches and tests within the run.

Running Tests through the Client

Tests are most easily and visibly run using the mvTest client.

The mvTest Client allows you to load and run a single test, a single batch or a complete test run, by right clicking an entry in the Explorer tree and selecting the **Run (Client)** option.

Before they are run the tests are first loaded into the Test Runner. If you are performing a test run, all the batches and tests in that run are cached before execution: this prevents loading times from interfering with tests, especially volume tests.



Test Run : MULTI_1   Sessions : 3

Session 0

Running EX2

```
Audio Book Catalog

 Title ID...:         18
 Main Title.:
[Hancock a Comedy Genius (BBC Radio Collection)    ]

 Author ID  : 18        Cost
 Reader ID  : 9         Cost

 ISBN......:  __________
 Format....:  CD

 Location:                          Stock Details

 Department:  ADULT                 Stock Level:          22
 Genre.....:  HUMOUR                Allocated..:           1
 Publisher.:  BBC Audiobooks        Price......:       12.99
```

```
Connecting to localhost
Connected
Waiting for name
Text Found
Sending brian
Waiting for word:
Text Found
Sending
Waiting for path
Text Found
Sending wychbooks
```

Script Runner

The test runner displays the progress of the test in three areas:

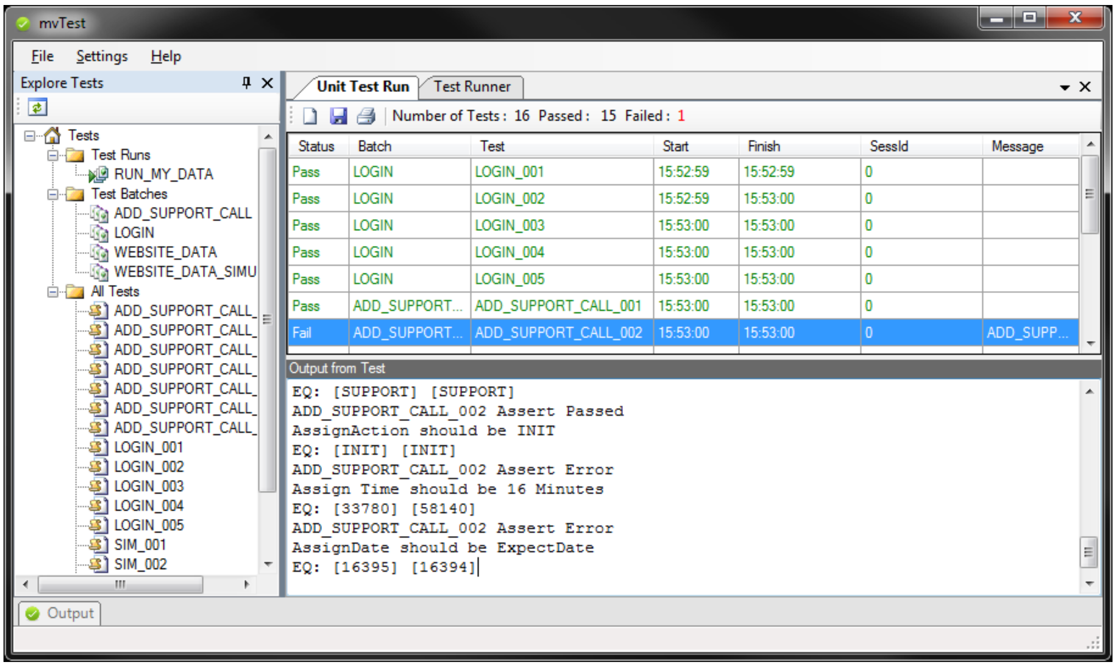
- A terminal window used to display text-based UI tests.
- A log of the test activities and their statii.
- An output panel for user defined messages from the test.

Click the Run button to invoke the test, batch or test run that has been loaded into the Runner.

The Test Results Window

The Test Results window shows the overall status of your tests.

This shows a single line recording the pass or failure of the test, and a panel below that records the activity involved in that test.



Number of Tests : 16 Passed : 15 Failed : 1

Status	Batch	Test	Start	Finish	SessId	Message
Pass	LOGIN	LOGIN_001	15:52:59	15:52:59	0	
Pass	LOGIN	LOGIN_002	15:52:59	15:53:00	0	
Pass	LOGIN	LOGIN_003	15:53:00	15:53:00	0	
Pass	LOGIN	LOGIN_004	15:53:00	15:53:00	0	
Pass	LOGIN	LOGIN_005	15:53:00	15:53:00	0	
Pass	ADD_SUPPORT...	ADD_SUPPORT_CALL_001	15:53:00	15:53:00	0	
Fail	ADD_SUPPORT...	ADD_SUPPORT_CALL_002	15:53:00	15:53:00	0	ADD_SUPP...

```
EQ: [SUPPORT] [SUPPORT]
ADD_SUPPORT_CALL_002 Assert Passed
AssignAction should be INIT
EQ: [INIT] [INIT]
ADD_SUPPORT_CALL_002 Assert Error
Assign Time should be 16 Minutes
EQ: [33780] [58140]
ADD_SUPPORT_CALL_002 Assert Error
AssignDate should be ExpectDate
EQ: [16395] [16394]
```

Tests are deemed to have failed if:

- an ERROR statement is encountered.
- an assertion has returned a false value.
- a UI test has timed out on a wait or expect condition.

The test results window provides a convenient place to quickly scan through a set of test results and to investigate any failures. The status bar at the top of the window records the total number of passes and fails.

Running Tests through the Server

The test scripts can be executed either client or server-side.

The client side scripting is the more visible and provides access to some features that are not supported on the server, such as running Windows commands. The client script engine should also be used for UI based volume testing.

But for most unit testing and regression testing scripts, the server side script engine provides a good alternative that can be run unattended if required directly from TCL or through a phantom.

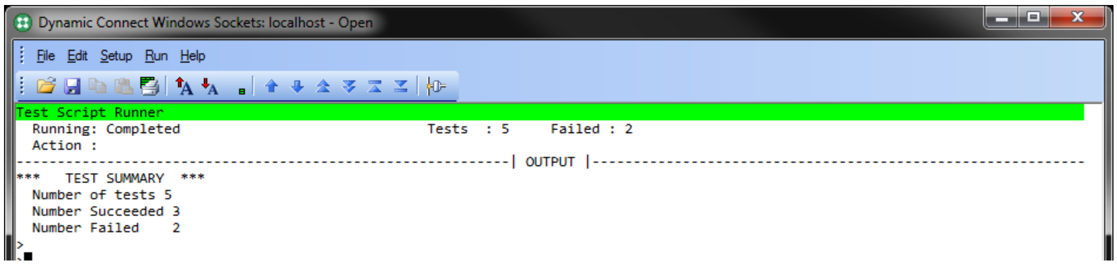
You can run server side tests in two ways:

- From the client by selecting the test, batch or run in the Explorer and clicking the Run Server option on the context menu. This runs the test directly in the server script engine but displays the results in the Test Results window (above).
- From the server by calling the test engine from TCL. This records the test results in the TEST_RESULTS file for later viewing and is the recommended way to perform regression and continuous integration (CI) tests. At the end of the run it can call a nominated subroutine to send or log notifications of the results.

The following command can be used at TCL:

TESTRUN {SCRIPT|BATCH|RUN} id [id..] options

This runs a test run or batch through whilst logging the results, or a single script without logging. You can also control whether notifications are sent and how the results are presented at the end. This should be used for live testing.



```
Test Script Runner
  Running: Completed                           Tests  : 5     Failed : 2
  Action :
-------------------------------------------------------------| OUTPUT |----------------------------------------------------------
***   TEST SUMMARY  ***
  Number of tests 5
  Number Succeeded 3
  Number Failed    2
>
```

Server side test runner.

Running a User Interface Test

The User Interface Tests are created with two scenarios in mind: regular testing for Regression or Continuous Integration, and specifically to support volume testing.

The User Interface Tests are designed to work with text based applications, including SB+ running in terminal mode. The interface tests allow you to script activities that represent the normal activities of a user logging in through a terminal emulator and making full use of the application.

At the same time, it allows you to include spot checks on the progress and to make decisions based on the responses of the application: handling pop-up selections or messages, testing the values that appear on screen and checking back to the database to ensure that the front end actions have resulted in the correct updates to the back end data.

You can also record your scripts though the UI Recorder (below) to make it easier to generate these in the first instance, and then edit them for general use.

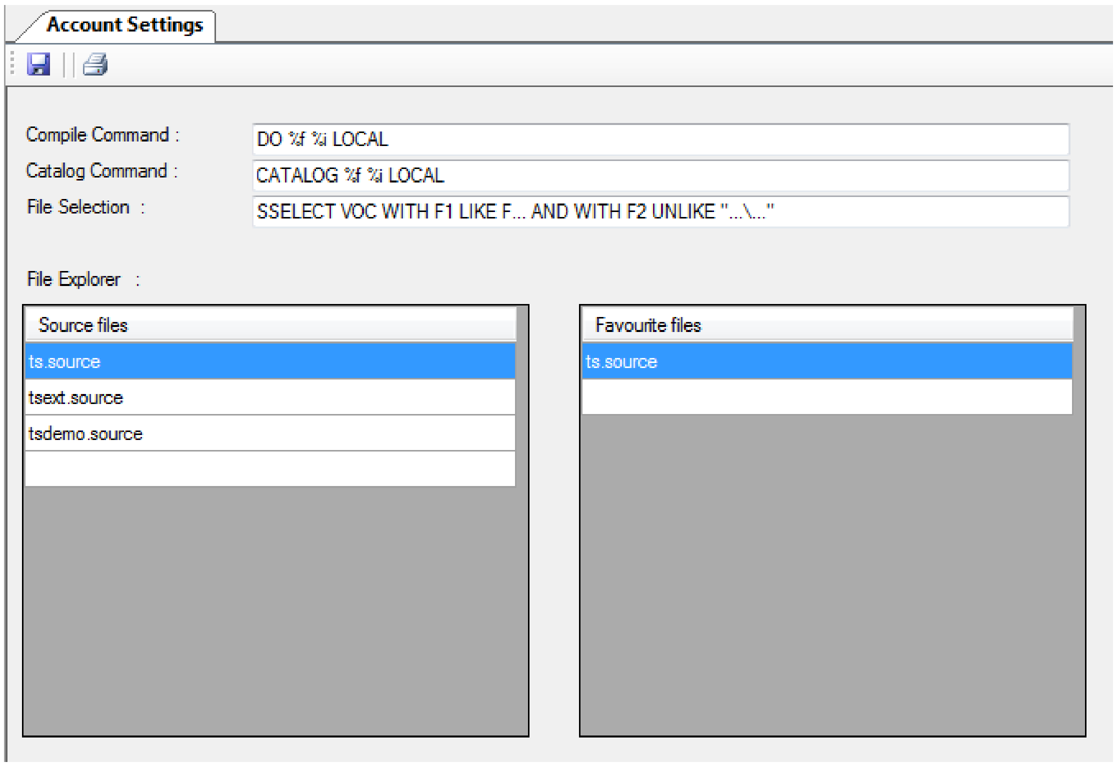
Using the Basic Editor

mvTest is designed to support Test Driven Development or TDD. This involves working with both code (system under test) and tests, often switching between them in very fast red-green-refactor cycles.

To make this easier for developers, mvTest itself includes a very capable editor for creating and amending UniBasic programs.

Configuring the Editor

The first step before using the Editor may be to configure your account settings for best operation. You can do so by selecting the Account option from the Settings menu:



Compile Command : DO %f %i LOCAL
Catalog Command : CATALOG %f %i LOCAL
File Selection : SSELECT VOC WITH F1 LIKE F... AND WITH F2 UNLIKE "...\..."

Source files
ts.source
tsext.source
tsdemo.source

Favourite files
ts.source

Compile and Catalog Commands

The account settings let you define the command used when compiling or cataloguing UniBasic code from within the editor. By default this will run a regular BASIC and CATALOG command, but you can override these if you have a specific catalog scheme or you are using a precompiler, or you wish to apply specific options to the commands.

The commands contain placeholders for the file name and item name being processed: %f for the filename and %i for the item name.

File Explorer

The File Explorer view presents a tree of the files in your account from which you can easily access items for editing or viewing.

If you have a large account, with many temporary files or files that are not required for development, this can take a long time to load and can be difficult to navigate.

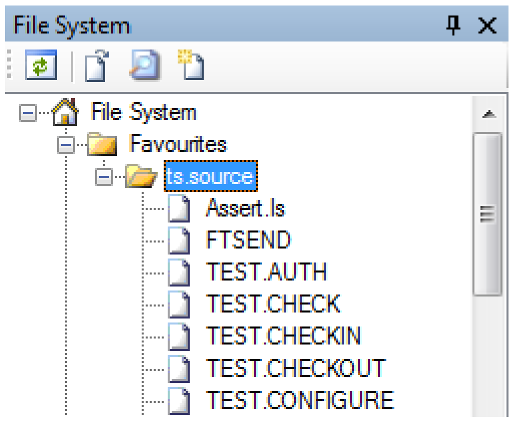
The File Selection command lets you specify your own command for selecting the list of files to appear in the All Files section of the explorer. This may be a regular selection with custom filtering expressions, or a QSELECT of an item holding a shortlist of interesting files.

The Source Files and Favourites lists fill out two branches at the top of the explorer tree for ease of access. mvTest cannot automatically identify source files, as site conventions for these change especially for sites that pre-process source code (i.e. it cannot just use .O files on UniVerse to identify source code).

Viewing the File Explorer

The File Explorer is not normally displayed on connecting to an account, as the time taken to retrieve the list of files in some accounts with large numbers of files and/or badly sized VOC files may be an annoyance.

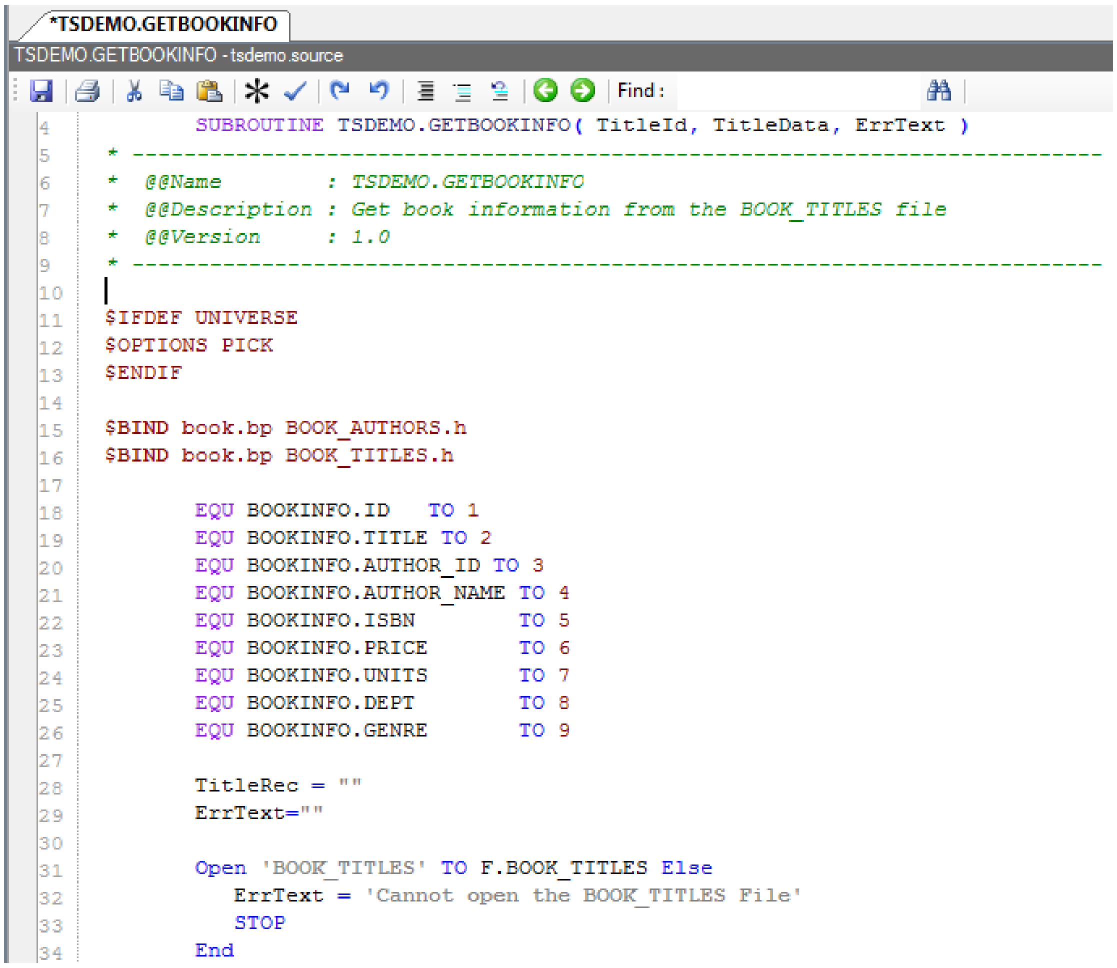
Instead you must explicitly open the File Explorer from the View Menu:



From here you can double click an item to load it into the Basic editor, or use the buttons on the toolbar to edit, view and create new items.

Using the Editor

The UniBasic editor is a fully featured editor with a number of special short cut commands for working explicitly with UniBasic source code and highlighting for UniVerse code.



```
         SUBROUTINE TSDEMO.GETBOOKINFO( TitleId, TitleData, ErrText )
* ---------------------------------------------------------------------------
*  @@Name          : TSDEMO.GETBOOKINFO
*  @@Description : Get book information from the BOOK_TITLES file
*  @@Version      : 1.0
* ---------------------------------------------------------------------------

$IFDEF UNIVERSE
$OPTIONS PICK
$ENDIF

$BIND book.bp BOOK_AUTHORS.h
$BIND book.bp BOOK_TITLES.h

       EQU BOOKINFO.ID   TO 1
       EQU BOOKINFO.TITLE TO 2
       EQU BOOKINFO.AUTHOR_ID TO 3
       EQU BOOKINFO.AUTHOR_NAME TO 4
       EQU BOOKINFO.ISBN        TO 5
       EQU BOOKINFO.PRICE       TO 6
       EQU BOOKINFO.UNITS       TO 7
       EQU BOOKINFO.DEPT        TO 8
       EQU BOOKINFO.GENRE       TO 9

       TitleRec = ""
       ErrText=""

       Open 'BOOK_TITLES' TO F.BOOK_TITLES Else
          ErrText = 'Cannot open the BOOK_TITLES File'
          STOP
       End
```

Here are some of the useful shortcuts:

F2	Save
F3	Search
F7	Format
F8	Compile
Shift-F8	Catalog
Ctl-A	Select All
Ctl-C	Copy
Ctl-D	Delete Selection
Ctl-G	Go to line
Ctl-L	Go to label
Ctl-N	Next Label
Ctl-P	Previous Label
Ctl-S	Save
Ctl-V	Paste
Ctl-X	Cut
Ctl-Y	Redo
Ctl-Z	Undo

Other shortcuts include Open Word at Cursor that can be used to quickly open called subroutines, Open Include at Cursor that opens files identified by a $INCLUDE or $BIND statement, jump to label from a GoSub, comment in and out blocks and similar shortcuts.

TIP

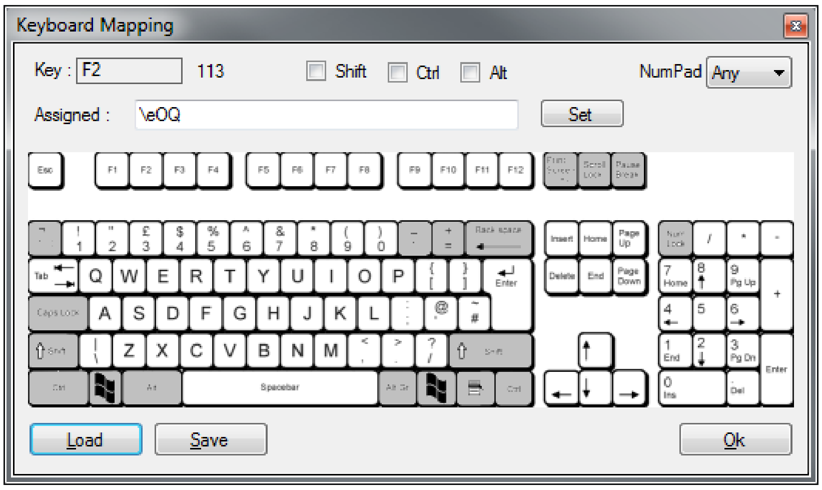
Remember that the docking within mvTest lets you move your windows around. If you are performing TDD it is best to have both the test and test subject open side by side.

Test Terminal

The user interface tests are supported in the mvTest client by a fully featured VT100 terminal.

This opens in the Test Runner for each session (see Volume Testing) and provides the means through which your scripts can interoperate with your application, including the use of function keys, support for video effects and screen capture.

mvTest comes with a standard vt100 definition, keyboard and colour scheme. You can override those by using the Settings menu to define new keyboard layouts and colours:

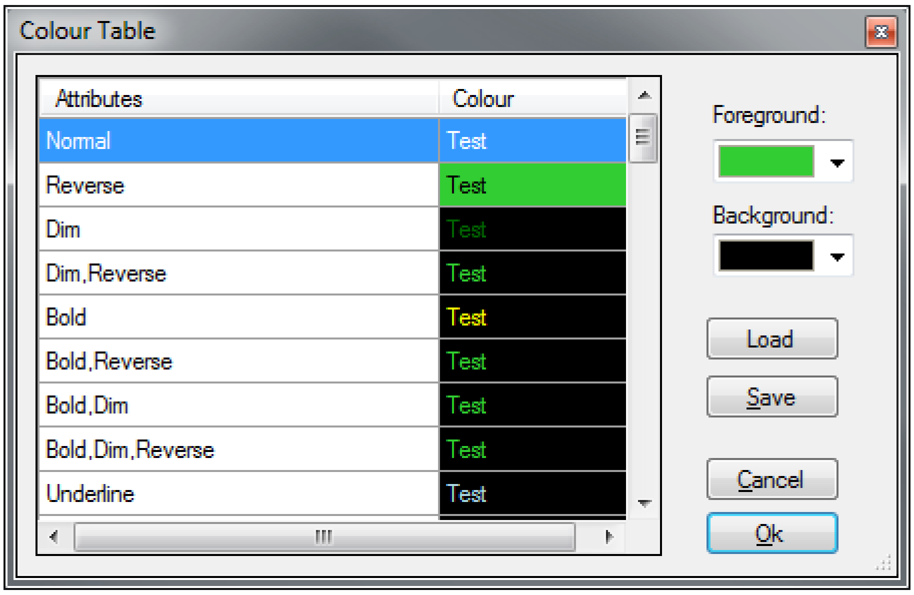


Keyboard Mapping

Key : F2 113 Shift Ctrl Alt NumPad Any

Assigned : \eOQ

Keyboard definition



Colour Table

Attributes	Colour
Normal	Test
Reverse	Test
Dim	Test
Dim,Reverse	Test
Bold	Test
Bold,Reverse	Test
Bold,Dim	Test
Bold,Dim,Reverse	Test
Underline	Test

Foreground: 🟩

Background: ⬛

Colour table

Example UI Test Script

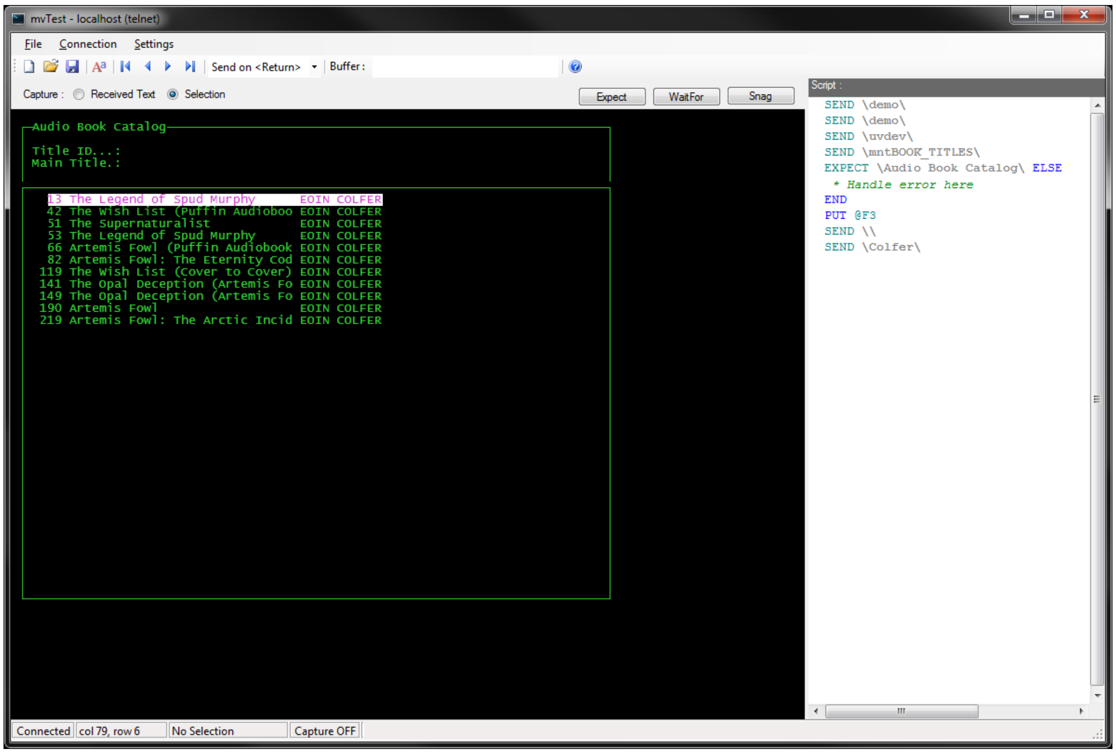
The following script provides a simple example of a user interface test: some details have been removed for simplicity and to protect the client.

```
SourceName = "TESTDATA"
Source = "c:\sources\sales_order.csv"
TestSource = 0

Runner.LoadCSV(SourceName, Source, TestSource)

IF TestSource = 0 Then
   Error "No test source created"
   STOP
END

* get the next set of test data unless we have come to the end
TestSource.Next( TestData )

If TestData = "" Then
   Error "No test data"
   STOP
End


Branch = TestData<1>
* LINES REMOVED FOR CLARITY
* populate test data


* Scripting Section
$INCLUDE TEST_SCRIPTS LOGIN

* Start SB+ and wait for the heading to appear
WaitUntil (Index(Snag(0,0,80),"MAIN MENU",1 ) > 0)

* Sales orders menu
Put "S"
Nap 100

* Create order
WaitFor "Create Order"
Send ""
Nap 100

* Wait for the Sales Order Entry screen to appear
WaitFor "Create Sales Orders"

* Branch
Send Branch
Nap 100

* Rep
Send Rep
Nap 100

* check for rep warning
WarningText = "Warning! - Rep is NOT for this Branch"
Screen = @SCREEN
```

```
If Index(Screen, WarningText, 1) Then
   Send ""
End

* Surname
WaitFor "Surname"
Send Surname
Nap 100

* Forename
Send Forename
Nap 100

* LINES REMOVED FOR CLARITY

*  ORDER LINES
WaitFor "Order Details"

For D = 1 To NoDetails
   Line = Raise(Details<D>)

   Type = Line<1>
   Product = Line<2>
   Qty = Line<3>
   Price = 0
   Client.OConvPrice(Line<4>,Price)

   * entry type
   Send Type
   Nap 100

   * product code
   Send Product
   Nap 100

   * Quantity
   Send Qty
   Nap 100

   * price is displayed but can be overridden
   Send Price
   Nap 100
Next

* Save screen
Put @F2

* Wait for order number
WaitFor "Order Number"
OrderNumber = Snag(38,22,8)
Crt "Order number ":OrderNumber


Put "/"
WaitFor "Process"
SEND "OFF"
Nap 100

Disconnect
```

Using the User Interface Recorder

Typing in user interface tests by hand is tricky and time consuming, not to mention error prone and somewhat tedious. So to assist you in building up your user interface scripts mvTest includes a UI test recorder. This embeds the same terminal used by the client and allows you to build up your script through capturing your keystrokes and spot checking the results whilst running your system normally.



```
Audio Book Catalog

Title ID...:
Main Title.:

 13 The Legend of Spud Murphy       EOIN COLFER
 42 The Wish List (Puffin Audioboo EOIN COLFER
 51 The Supernaturalist            EOIN COLFER
 53 The Legend of Spud Murphy       EOIN COLFER
 66 Artemis Fowl (Puffin Audiobook EOIN COLFER
 82 Artemis Fowl: The Eternity Cod EOIN COLFER
119 The Wish List (Cover to Cover) EOIN COLFER
141 The Opal Deception (Artemis Fo EOIN COLFER
149 The Opal Deception (Artemis Fo EOIN COLFER
190 Artemis Fowl                   EOIN COLFER
219 Artemis Fowl: The Arctic Incid EOIN COLFER
```

```
SEND \demo\
SEND \demo\
SEND \uvdev\
SEND \mntBOOK_TITLES\
EXPECT \Audio Book Catalog\ ELSE
 * Handle error here
END
PUT @F3
SEND \\
SEND \Colfer\
```

The UI Test Recorder generates a script of your activities as you run your application. By default this captures each line you type (terminated by a <Return>) and any special keys such as function keys, cursor keys or control keys. If it is more applicable to your application, you can also tell it to capture each key individually by changing the Capture option on the toolbar to **Put each character**.

As you build up the script, you will need to 'spot check' where you are in the application by expecting or waiting for certain things to appear on screen. The UI recorder does not do this automatically or the resulting script becomes too long and ugly and edit easily. Instead at key points where you want to react, such as a menu appearing or a screen header being displayed, you can click the Expect or WaitFor buttons and it will add the latest text received to the script.

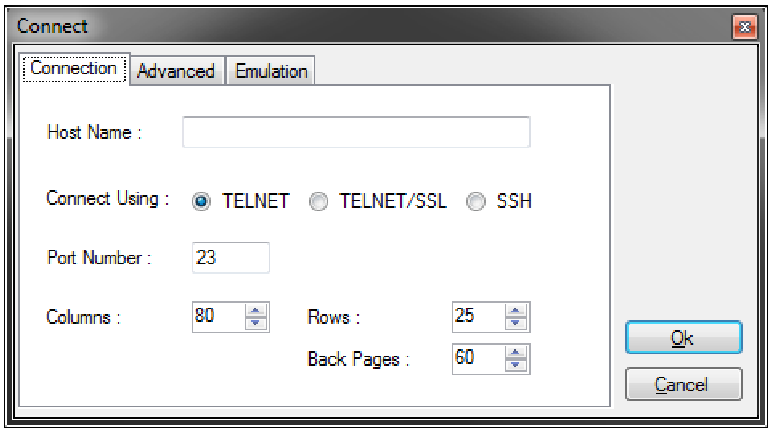
For more precision, you can also select part of the screen and click the Expect or WaitFor button. This will add the text at that point as the value to expect, and is recommended in most cases. Similarly, if you want to tell mvTest to snag an area of the screen for testing, you can highlight that area and click the Snag button. That will add a line to the script to snag that area.

Once your script has been recorded in this way you can save it to a local text file. It is very unlikely that you would run the same test in exactly the same way though mvTest, so this allows you to edit the script before pasting it into the mvTest client.

You can still edit the script as you go along, adding comments or refining what the UI recorder has captured.

Connecting to the Database

To connect to your database server, click the Connect option or open a stored connection from the Connection menu. This gives you connection options for the terminal layout, host to which to connect and the transport type: the UI Test Recorder supports TELNET, TELNET/SSL and SSH connections.



Host Name :

Connect Using : [✓ TELNET] TELNET/SSL SSH

Port Number : 23

Columns : 80

Rows : 25

Back Pages : 60

TIP: Remember to set the number of rows and columns to your regular terminal display.

Setting up the Keyboard and Colours

The mvTest UI Recorder is a fully functioning terminal emulator, and allows you to define your keyboard macros and colour scheme in the same way as the mvTest client, as described earlier in this user guide.

You can save your connection details, including the colour and keyboard mappings, to a stored connection profile. These are held in the Documents\mvTest folder on your PC so that they can be easily copied and shared between different testers.

Volume Testing

Volume Testing is an extension to the regular testing that allows you to multiple test run sessions, particularly with regard to user interface testing.

With Volume Testing, the Test Run defines the number of sessions, number of iterations of the full test and the pauses on start-up and between each run to spread the load more accurately. You can also decide to randomize the batches that make up the test run.

Each test is performed through a separate instance of the test engine and on a separate thread. This changes the way in which the terminal operates to a polling mode that is less immediate in its feedback but uses lower resources at the client.

For user interface testing, each test runner creates its own TELNET connection but can share a single control connection for direct database access. This prevents the control connections from swamping the database or using excessive number of user licences, leaving that to the telnet sessions being tested.

Hint:

OCONV and ICONV make return trips to the server. You should avoid using these in your tests. Use the CLIENT methods to perform conversions instead where possible.

Using the Session Id

Each session is identified by a session number running from zero. This is available from the @SESSIONID variable.

The session id can be used to disambiguate tests and also to ascribe certain tests to specific sessions or subsets of the full range of sessions running.

For example, if you have an activity that would normally only be performed by a small number of users whilst other activities are performed by all users, you can use the session id to restrict the more unusual activities to particular sessions. A tip is to use Test Run variables to hold those restrictions.

See the ABORT statement for more details.

High Volume Tests

Eventually with higher volumes, the effect of running large numbers of sessions in parallel may affect the client, or the delays on the client may affect the running time of the tests and thus influence the results.

To further reduce the overheads, you can run user interface tests in hidden mode. This continues to use the terminal as a source so that tests on the @SCREEN, buffering and snagging areas will continue to operate, but the terminal will not display its activity.

Integration Tests

Integration tests can be performed using the client runner, but if possible these are better managed through the server side runner. This allows you to fire off tests as phantoms and record the results, making it possible to do continuous integration testing.

For server side testing, you can optionally specify a Notification Routine as part of the test run definition. This is the name of a catalogued Basic subroutine that can you provide yourselves to fit into whatever workflow or email system you use at your site. The notification routine expects the following standard call parameters:

SUBROUTINE Notification(InData,OutData,ErrText)

The InData parameter contains:

Field	Meaning
1	Key to the record in the TEST_RESULTS file holding the detailed results of the run.
2	Name of the TEST_RUN definition that has been run.
3	Total number of tests run
4	Total number of tests passed
5	Total number of tests failed

You might, for example, use this routine to email a warning should the number of fails be greater than zero.

Note that running continuously is not the best option for CI testing. If your database caches subroutines in memory until returning to the command level, as in the case of the Rocket UniVerse database, a continuous test run will not pick up any changes to those subroutines as they get loaded to the test system. It is better in those circumstances to phantom the test runner on each software load.

CHAPTER 2 WRITING TESTS

Creating Tests

The art of automated testing relies on being able to create simple but effective unit tests, and to combine these to perform system testing with adequate, or otherwise predictable, coverage. If you have a legacy application it is unlikely that your first port of call will be to revisit every part of your system building unit tests, so in this case it is more important to tie testing to releases and to change requests, working in small blocks so that you know which areas have or have not been covered by your testing.

In this chapter we will look at how unit tests are created using mvTest. These follow the Samples installed in the mvTest account. Please note that for some of these you will need to edit the scripts to match your installation.

Where to Create Your Tests

One of the first issues to address when performing testing, especially where there are multiple teams or several developers working on the same project, is where the tests should be sited in order to run and share them.

Running tests ideally takes place in a clean environment where you can control your data. That may mean creating a new account in which to run your tests where you have no test data set up to begin, or running your tests in an existing test account where you can easily check the before and after state of your data. If you are a creating a sales order, you can easily check that in isolation. If you are running a report, you may need to be more careful about pre-populating your files.

In either case, if there are many developers collaborating and running tests, you may need to ensure that they do not step on each others' toes. For example, if you want to clear a file and populate it with your test data for a report, you don't want a colleague running tests that add their own data to that file in the middle of your test!

mvTest lets you set up your testing accounts in one of three ways:

- using local tests
- using a shared test repository
- using source control

Using Local Tests

The default for mvTest is to create local files to hold all your test scripts and results. This is the best option if you have a dedicated test account and your developers are not going to be treading on each others' toes during testing.

Using Shared Test Repository

When you enable an account using the TEST.SETUP command, it will prompt you for the location (optional) of a test repository. This is a separate shared account where all your tests will live, and enables you to set up individual accounts in which to run your tests, whilst still keeping the tests centrally so they can be shared.

This is the best option if you have multiple developers each with their own test accounts.

Using Source Control

mvTest has a simplified source control API built into the mvTest client. When enabled, this lets you navigate a repository of tests under your preferred source control system. Your developers can then use local files for their tests, but submit and pull them to and from a central repository. More details on this can be found in the mvTest Source Code Control User Guide.

Using Assertions

(See Samples 100 – Assertions, 101 – Assertion Fails)

Assertions lie at the heart of unit testing.

At its base, an assertion is a test for expected functionality. mvTest has a number of different assertion types, but all of these effectively do the same thing: they test for an expected outcome and report a FALSE value for the test if the outcome is not met.

In mvTest, all feedback is handled by assertions. A test is considered to PASS if there are none of its assertions have failed.

The basic assertion syntax is:

Assert message condition

The message allows you to track which assertion failed, and is recorded in the Test Results window (client) or Test Results file (server). A test can contain multiple assertions.

For example:

```
Assert "Today should be a good day", Feeling = "Good"
```

The other assertion statements all provide specific tests to simplify writing your assertion code and to make the scripts more intelligible. For a full list see Chapter 3 (Scripting).

The **AssertIs** and **AssertNot** statements test for equality and inequality respectively:

```
AssertIs "Today should be a good day", Feeling, "Good"
AssertNot "Today should not be lousy", Feeling, "Lousy"
```

Similarly, **AssertHas** and **AssertHasNot** tests a dynamic array to see whether it includes a specific element. The assertion checks at the highest level of the array provided, so if the array is multivalued it will check each value, if not it will check each field.

```
Line = "SUNSHINE":@FM:"OF":@FM:"YOUR":@FM:"SMILE"
AssertHas "Today should be good", Line, "SMILE"
AssertHasNot "No frowns please", Line, "FROWN"
```

The **AssertEmpty** and **AssertFull** assertions simply test whether an expression contains a value or not.

```
AssertEmpty "There should be no errors", Errors
AssertFull "There should be a record", Record
```

Calling Subroutines

(See Samples 200 – Calling Subroutine)

The best structured MultiValue applications are easy to test.

In a well structured application, the designer will already have separate out the business logic from the presentation, making it easier to develop with different front ends and enabling good code reuse. If you are fortunate enough to be in that situation, most of your unit tests can consist of calling Basic subroutines with predictable or randomized test data (more later) and testing for the expected results.

In fact, in the very best systems, you may already have decided on a calling convention to follow that promotes the use of automated testing tools like mvTest and single point calls from a front end through a dispatcher to facilitate ease of logging, auditing and debugging.

Of course such things are not always possible, especially if your system has been developed using legacy 4GLs or you have inherited a system that has grown organically through many hands with no concern for future-proofing. Even in these cases it is worth looking at the benefit of refactoring key pieces of business logic into external subroutines as part of your daily change operations.

There are some limitations that are unfortunately imposed by the middleware used:

- No dimensioned arrays.
- No special variables.

If you need to pass dimensioned arrays, create a stub routine to call and use MATBUILD/MATPARSE in Basic.

Calling a Basic subroutine in mvTest is similar to Basic: it uses the CALL statement.

Call subroutine_name (args)
Or
SubrName = "MY.SUB"
CALL @SubrName(args)

For example:

```
CustId = "TEST1"
Expected = 10000

InData = CustId
OutData = ''
ErrText = ''

Call CreditLimit(InData, OutData, ErrText)

AssertEmpty "ErrText should be empty", ErrText
AssertIs "OutData should be ": Expected, OutData, Expected
```

(See Samples 201 – Reading Data)

One of the benefits of mvTest is that it is fully integrated into the database for all types of tests, including UI testing (covered later). This means that, for example, you can run an entry screen filling in details then immediately check that the screen has saved the data in the correct format.

The mvTest scripts support the Basic style of OPEN, READ, WRITE and DELETE statements:

```
Open 'CUSTOMERS' To CUSTOMERS Else
  ERROR "Cannot open CUSTOMERS"
  STOP
End

Read CustRec From CUSTOMERS, CustId Else
  CustRec = ''
End
```

Note that the ERROR statement also causes a script to fail.

Because scripts so often test for data, there is a short-cut function for reading:

Record = Read(filename, recordId)

For example:

```
* first check the demo database has been installed
Temp = Read("VOC", "BOOK_TITLES")
AssertFull "Demonstration database installed", Temp
```

Checking Values

(See Samples
For testing more complex data or return values you can use the CHECK() function.

Result = Check(Value, Conditions)

This gives a short hand series of validation conditions expressed as a string, which could be held in a separate file. These include range tests, lists of possible values, related keys and pattern matches.

For example:

```
* Author should be an integer and a key on the BOOK_AUTHORS File
AuthorId = TitleRec<2>
Condition = "T:I;R:BOOK_AUTHORS"
Assert "Author Id should pass tests", CHECK(AuthorId, Condition)
```

The checks can be assembled from the following conditions:

C	code	Checks value against a conversion code
F	Filename	Checks that the value is a key to a record on filename
L	length	Checks the length of value
P	Pattern	Checks value against a pattern match
R	low-high	Checks value against a range
T	N,I,T,D	Checks value type is a number, integer, date or time
V	list	checks value against a comma separated list

Of course, you can also use individual functions such as NUM() and LEN() as you would in Basic. The txext.bp file holds some additional special checks as subroutines to validate specific types of data, such as valid credit card numbers and ISIN codes.

See Samples 203 – JSON DATA, 205 – JSONPATH DATA

The data handling is based around the MultiValue with support for regular dynamic array operations. Sometimes, however, you may need to test return data in other formats where you have generated this for external consumption, such as XML or JSON.

The JSON() function returns a specific keyed value from a JSON dictionary:

Value = JSON(Dictionary, Key)

For example:

```
JSONData = \{'id':'190','title':'Artemis Fowl','author_name':'Eoin Colfer'}\

AuthorName = JSON(JSONData, 'author_name')
```

The Client Side Runner also supports the JSONPath syntax. This allows you to make more complex data selections against JSON formatted data, similar to the XPATH syntax for XML.

```
{ "store": { "book": [ { "category": "reference",
                    "author": "Nigel Rees",
                    "title": "Sayings of the Century",
                    "price": 8.95
                },
                { "category": "fiction",
                    "author": "Evelyn Waugh",
                    "title": "Sword of Honour",
                    "price": 12.99
                },
                { "category": "fiction",
                    "author": "Herman Melville",
                    "title": "Moby Dick",
                    "isbn": "0-553-21311-3",
                    "price": 8.99
                },
                { "category": "fiction",
                    "author": "J. R. R. Tolkien",
                    "title": "The Lord of the Rings",
                    "isbn": "0-395-19395-8",
                    "price": 22.99
                }]
            }
        }
```

```
Expr = "$.store.book[*].author"

Result = JSONPath( Temp, Expr )
AssertFull "JSON Path should find authors", Result
Expected = "Nigel Rees"
AssertIs "First author should be ":Expected, Expected, Result<1>
```

Using Randomized Data

(See Samples 204 – Random Data)

Working with predictable test data enables you to test expected functionality, but offers little scope for performance testing or catching unexpected behaviours. So mvTest has a number of different ways in which you can generate test cases using randomly selected entries from a source.

The first of these is the ONE() function. This takes a specification of possible values from which to generate a randomly selected sample:

Value = ONE(specification)

The specification is similar to that used for the CHECK function:

F	Filename	Randomly select a key from filename
R	low-high	Randomly select a key from a range
V	list	Randomly select from a delimited list of options

For example:

```
* Select one of the first 250 titles
Specification = "R:1-250"
TitleId = ONE(Specification)
Assert "Title should be between 1 and 250",
CHECK(Specification,TitleId)
```

Using Data Sources

(See Samples 300 – CSV Data Source. You will need to modify the paths for your setup).

A more powerful means of populating a test, especially designed for UI testing, is the use of external data sources. These are typically comma separated files containing lists of potential related test data, which are loaded by the runner client and returned in either a static or randomized order.

A data source is loaded as a client object using the Load.. methods of the static RUNNER object. The initial release of mvTest only supports delimited data sources, however additional data sources may be provided in later revisions based on demand.

A CSV Data Source loads a delimited file into memory and provides access to the data in the file at a row or cell basis. The Data Source assumes that the first line of data contains the column headers: these are used to populate the data source.

Each data source is associated with a name. This name is used internally by the mvTest client to identify an instance of the data source, so that if you are running volume tests you can share the same test data between multiple runners if you desire.

The data source exposes a number of methods for accessing data including:

For example:

```
Source = RUNNER.LoadCSV("TITLES_CSV", "c:\temp\titles.csv")
Assert "Source should not be empty", Source > 0

* Shuffle the pack
Source.Shuffle

* Get the current (first) title
Title = Source.Get("Title")
AssertFull "Title should not be empty", Title

* Get the next row
NextRecord = Source.Next
AssertFull "Next record should not be empty", NextRecord

* All done with the test source
Source.Close
```

Writing UI Tests

(See Sample 400 – Login and Disconnect)

User interface tests provide a mechanism for driving a terminal based application. These can range from simple send and expect operations to more complex checks for data displayed and handling potential error messages.

When running a UI test, the runner initiates two connections to the database: the regular test connection used to handle read, call and similar operations for the script, and a terminal based connection over TELNET, SSL or SSH.

You will need to create a script to navigate through your login to the account. This typically uses the @USER and PASSWORD primitives to reflect the user name and password with which you logged into mvTest – this prevents storing credentials in your script.

It is a good idea to create a login script as a separate test then $INCLUDE it into your User Interface tests.

The primitive statements for UI tests are:

CONNECT	opens a terminal connection to the server.
DISCONNECT	closes the connection
WAITFOR text	waits for text to appear in the terminal stream.
SEND text	Sends text followed by a line break.
PUT text	Sends text with no terminating line break.
PASSWORD	Sends the current user password.
WAIT n	Sleeps for n seconds

For example (connecting to UniVerse on Windows):

```
CONNECT
WAITFOR "name"
WAIT 1
SEND @USER
WAITFOR "word:"
WAIT 1
PASSWORD
WAITFOR "path"
SEND "MY_ACCOUNT"
WAITFOR "logged on"
SEND "TERM vt100"
```

Creating a Secure Connection

(See Sample 401 Connection over SSL)

mvTest supports the regular TELNET protocol, but also handles secure connections. The first revision supports the UniVerse and UniData TELNET over SSL capabilities. Later revisions will also support SSH.

You can find instructions and utilities for setting up the TELNET/SSL feature for a UniVerse or UniData database on my blog, available at www.brianleach.co.uk.

The connection type is set through the writeable @TRANSPORT script system variable. This can be set to:

- telnet
- ssl
- ssh*

For example:

```
@TRANSPORT = "ssl"
```

(See Samples 402 – Simple Screen Update, 403 – Screen Capture)

Most UI tests perform some level of screen or menu driving. To understand this, you need to understand how mvTest interprets screen activity.

The mvTest screen runner is similar to a terminal emulator but works by polling the source for data unlike the regular asynchronous socket operations used for screen handling. This makes it more efficient when used with large numbers of connections. The terminal maintains a buffer of current data, and the screen runner can interpret the stream of data coming back from the server raw as well as using the terminal layouts.

To watch screen activity you can use the buffer with WAITFOR or EXPECT statements, which simply watch the incoming data until the expected text has been received. WaitFor will time out after 60 seconds if the text has not been found, and terminate the script. Expect also has the same timeout but allows you to take your own actions through the Else clause.

WaitFor text
Expect text Else …

The Simple Screen Update sample uses this to change the data on a record through an entry screen and check the results.

You can also test the @REPLY variable. This waits to grab the next response from the server, so can be used to check, for example, whether the server has responded to a command that has been SENT:

SEND "WHO"
Response = @REPLY
RemotePort = OConv(Response,"MCN")

You can access the whole screen image through the @SCREEN variable. This holds the screen as a dynamic array of rows:

```
Text = @SCREEN
If Index(Text,"This record has been locked", 1) Then
   Crt "Record is locked"
   Send ""
   GoSub Finish
End
```

You can also snag areas of the screen using cursor locations, or access the whole screen image through the @SCREEN variable.

Image = SNAG(col, row, length)

For example:

```
Author = Trim(Snag(15, 6, 10))
Ok = Check(Author,"T:N;F:BOOK_AUTHORS")
AssertIs "Author should be valid author", Ok, @True
```

Some more powerful actions are provided by the **LookFor** function. This allows you to check for multiple sets of actions, for example where you are passing test data into a screen that could result in one of several validation messages, and **WaitUntil** that runs a test at intervals until a condition has been met.

WaitUntil condition

For example:

```
WaitUntil (@COL = 10)
```

The polling interval for these is normally set at 100ms on the client and on UniVerse. For the server test runner on UniData this expands to a second as UniData does not support sub-second NAP statements.

HINT

You can use the mvTest UI Test Recorder to run a terminal session and capture the keystrokes and terminal output into a script format. This takes the heavy lifting out of trying to build a UI script manually.

Suggestions for Writing Unit Tests

Writing Unit Tests is an art and a discipline that is very similar to the art and discipline of writing the underlying code. A unit test may be simple, but should be approached and designed with the same diligence as production code. There is no point having a large number of tests that achieve nothing other than instilling a sense of false confidence or for box ticking.

You will find various suggestions for how to write unit tests and how to work with legacy code on my blog, accessible from www.brianleach.co.uk. These include real world examples and lessons learned from the trenches.

A test script should describe the actions of the code being tested. In a very real sense, the unit tests should document what that code does and should be the first port of call for developers who are picking up that code for the first time.

A suggestion adopted from other test suites is to use a naming convention that expresses this, as follows:

```
! This describes the overall test
GoSub Initialize
GoSub ShouldDoSomething
GoSub ShouldDoSomethingElse
GoSub Teardown

ShouldDoSomething:
Test for that something
Return
..
```

For example, the following test is a part of the testing for the mvTest Profile Configuration (described below):

```
! Test the Configuration Creation and Teardown
$INCLUDE ts.source TEST_PROFILE.h

GoSub Initialize
GoSub ShouldCreateNewFile
GoSub ShouldOverrideExistingFile
GoSub ShouldCopyDicts
GoSub ShouldCreateNewPointer
...
GoSub Teardown

STOP
```

The naming convention makes it easier to focus on the actions you want to test – you can start with a piece of paper and just write down all the things the code should do. Within the test script each such unit test should be self contained and should test one action that the code being tested must perform.

Working with Test Profiles

Often you will need to work with predictable data for your unit testing to succeed.

Unit tests are designed to work with minimal data. You cannot depend on the data in your system for testing – if, for example, you take an end-of-day snapshot, that may not happen to contain the combinations you need if your data is fast-moving. Similarly, if your routine requires changes to the data structures introduced as part of the same release, these changes may not be present in your current test data. Your tests may change the data in a way that prevents them being run repeatedly. And unit tests are intended to be fast and run many times a day, so normal data volumes may just be too large.

For accurate unit testing, you need to create and destroy data as needed. This is probably the most time-consuming part of unit testing, which is where mocking of routines that depend on data and mocking of files to hold just the data you need (and no more) are very important to get you running quickly.

At the heart of many unit tests are the setup and teardown activities in which you may populate data files with specific test data, run your tests and then clear down the tests afterwards.

If you are working in a distributed environment in which every developer or team has a separate account in which to develop code, it is also important that the same tests can run in any such account, with no dependencies on files being in existence. This is particularly true when looking at continuous integration. You may need a way to make a development or CI account 'look like' a real data account. This is where the test profiles come in.

The Test Profiles provide one convenient means of ensuring that you can work safely with local data files. A Test Profile creates temporary files in your account at the start of a test batch and then clears down those files at the end. You can populate the file dictionaries from master files held in other accounts so that they are ready for performing any dictionary driven operations. You can even populate data, though this is really only suitable for static data or control files.

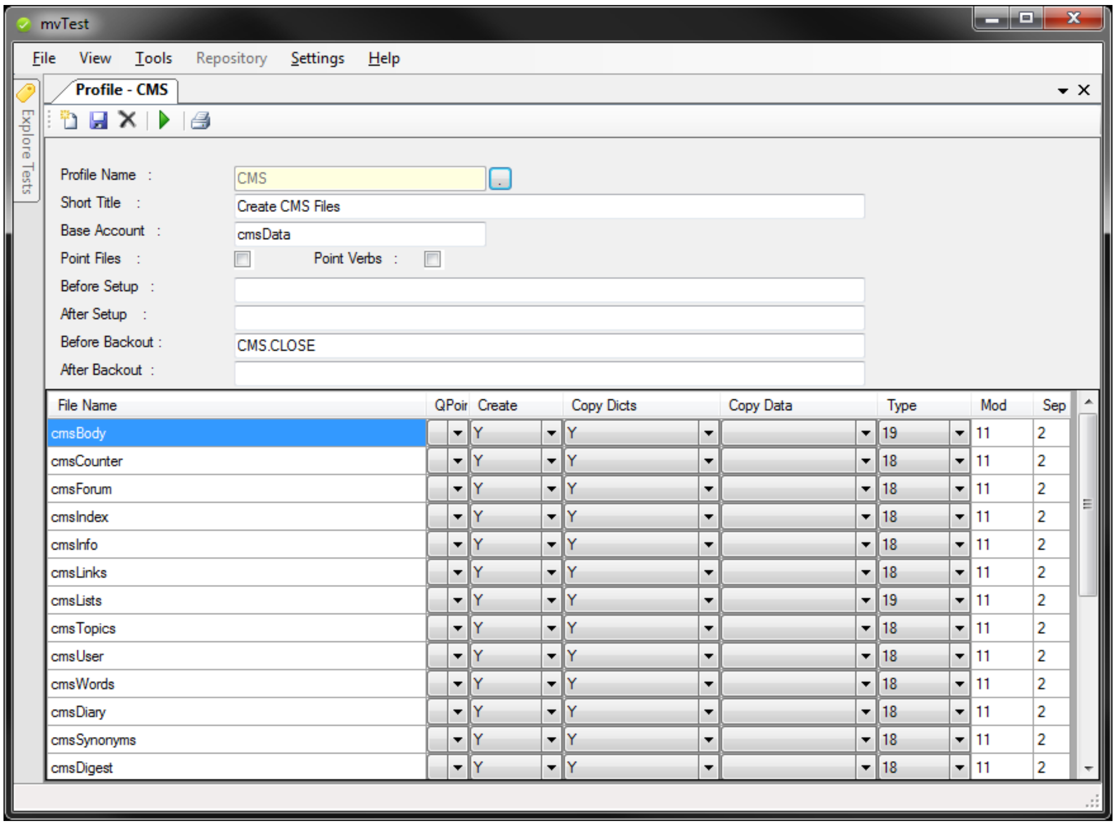
These files are physically created using temporary names and the file pointers in the VOC or MD are switched to point to these temporary files. At the end of the batch, the original pointers are restored and the temporary files deleted.

Warning:

UniVerse will not delete files that are held open in named common. Be careful to close any such files that are opened by your code under test.

Creating a Profile

Profiles can be created under the Tools menu in the mvTest client:



Profile Name :	CMS
Short Title :	Create CMS Files
Base Account :	cmsData
Point Files :	☐
Point Verbs :	☐
Before Setup :	
After Setup :	
Before Backout :	CMS.CLOSE
After Backout :	

File Name	QPoir	Create	Copy Dicts	Copy Data	Type	Mod	Sep
cmsBody		Y	Y		19	11	2
cmsCounter		Y	Y		18	11	2
cmsForum		Y	Y		18	11	2
cmsIndex		Y	Y		18	11	2
cmsInfo		Y	Y		18	11	2
cmsLinks		Y	Y		18	11	2
cmsLists		Y	Y		19	11	2
cmsTopics		Y	Y		18	11	2
cmsUser		Y	Y		18	11	2
cmsWords		Y	Y		18	11	2
cmsDiary		Y	Y		18	11	2
cmsSynonyms		Y	Y		18	11	2
cmsDigest		Y	Y		18	11	2

The profile defines the set of files to create or point to in a remote account. If the remote account is set, you can also copy the dictionaries and data for the remote file into the dictionary of the local file when it is created. If the original file contains secondary indices on its fields, these indices will also be automatically recreated on UniVerse.

The Point Verbs further allows you to create remote pointers to any verbs, sentences, paragraphs or PROCs in the VOC of the base account.

For each file you can specify the type and size of file to create. As you are only creating test data, these are often very much smaller than your live data files. If the files need to be pre-populated with known data, you can use the command After Setup to do this. There is also a Copy Data option to simplify creating local copies of control files or files with static data. This should not be used for your test data though as this can become too brittle.

Dealing with Files In Named Common

The command Before Backout is a useful placeholder for closing files opened in common before the backout commences.

UniVerse cannot delete files that are open in named common and so, whilst the mocked VOC pointers can be switched back to the original real files (if they exist), Universe will leave the temporary files behind to junk up your account. A simple routine here can save a lot of trouble, e.g.

```
COMMON /MYCOMMON/ MY_COMMON_FILES(MAX.FILES)

For ThisFile = 1 To MAX.FILES
  If FileInfo(MY_COMMON_FILES(ThisFile,0)) Then
     Close MY_COMMON_FILES(ThisFile)
  End
Next
```

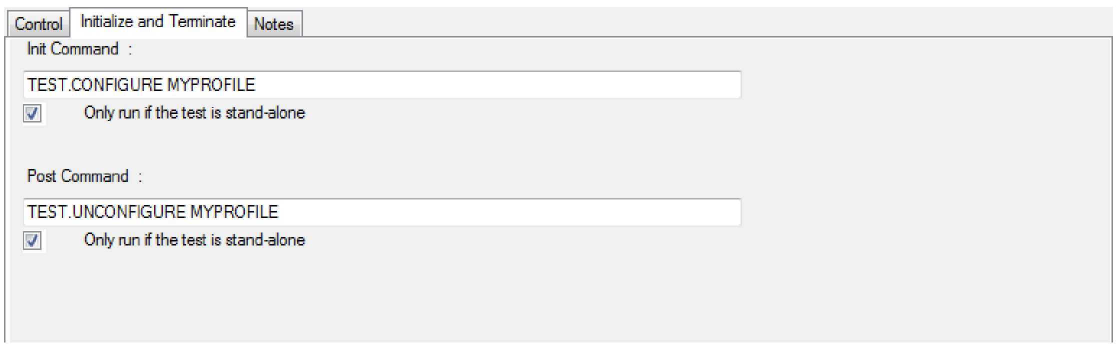
Running a Profile

You can run a profile from the command line by using the following commands:

TEST.CONFIGURE profile_name	Create the specified profile.
TEST.UNCONFIGURE profile_name	Destroy the specified profile.

You can run these from the Init Command and Post Command of a Test Batch or Test Run. You can also do the same for a Test Script, and can specify whether you only want to run this if the test script is stand-alone, so that you can safely apply the same profile to the batch in which it runs without recreating it for each test in the batch.

If you wish to split your configurations into several profiles for ease of use, you can specify a number of profiles on the command line for both TEST.CONFIGURE and TEST.UNCONFIGURE.



Init Command :

TEST.CONFIGURE MYPROFILE

☑ Only run if the test is stand-alone

Post Command :

TEST.UNCONFIGURE MYPROFILE

☑ Only run if the test is stand-alone

If a profile or profiles have been left behind, generally from a fatal error or other crash of the system under test, you can clean up any existing profiles using:

TEST.UNCONFIGURE.ALL.

If you want to run the configuration from your own set up and tear down routines, you can also direct call the tsConfigure subroutine with the following arguments:

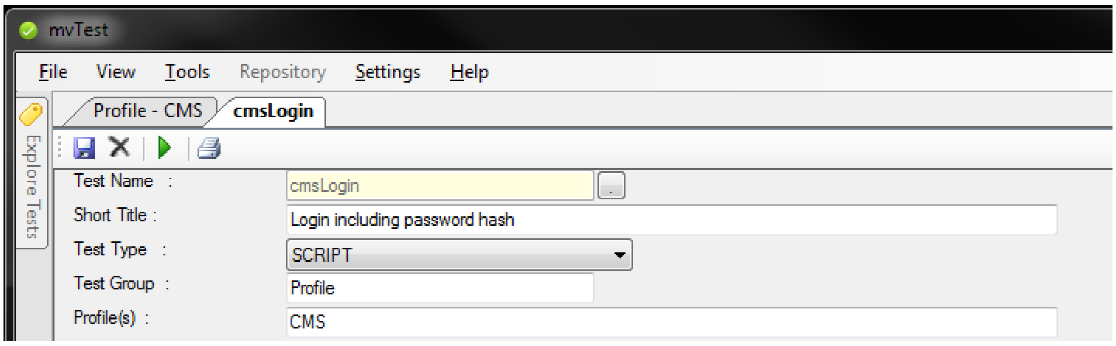
CALL tsConfigure(Action, ProfileName, OutData, ErrText)

Where Action is set to:

1. Create the profile.
2. Tear the profile down.
3. Tear down all existing profiles.

Associating a Profile with a Test

Once you are confident that your profile is working correctly, you can associate it directly with your tests to save populating the Command Before and Command After tests.



Test Name : cmsLogin

Short Title : Login including password hash

Test Type : SCRIPT

Test Group : Profile

Profile(s) : CMS

The Test Details view allows you to enter one or more profiles that should be active when the test runs. The configuration manager will automatically create the profile if it does not exist when the test is run.

If the test is running stand-alone the manger will clear down the profile after the test has completed.

If the test is running as part of a batch, the manager will persist profiles between tests if both tests have the same profile in their list. This allows you to maintain data that you need between tests in a batch, for example where the batch is part of a workflow that you want to run as an end-to-end test. You can, of course, explicitly clear down the profile if you need.

At the end of the batch, all active profiles attached to the tests are cleared down automatically. By design, you cannot persist these between batches in a test run.

CHAPTER 3 SCRIPTING

SCRIPTING LANGUAGE

The Script Language is a hybrid language designed to appeal to existing UniVerse, UniData and multivalue programmers whilst including specialized functions and language features.

The language is fundamentally syntactically modelled after UniVerse Basic. Unlike Windows languages the Scripting language is typeless, does not require variable declaration, handles dynamic arrays and generally follows the same structures as UniVerse Basic. This extends right through the scripting language: the same looping and branching constructs are supported that you will find in UniVerse – even the same Boolean evaluations.

However, whilst appearing similar the scripting language is **not** UniVerse Basic.

The scripting language is an interpreted (compiled) language, with the addition of functions specific to the host environment and the ability to access properties and methods of the tools and static objects provided. Thus it provides a convenient and easy half-way house between the UniVerse procedural language and a component-oriented Windows language.

Scripting Language Syntax

The scripting language is similar to UniVerse Basic, but there are some significant differences. Please ensure that you read through this section to ensure that you don’t get confused by these.

Language Layout and Primitive Elements:

Scripts are laid out as a series of statements, one statement per line. The script language does not support multiple statements per line as that is simply ugly. Blank lines are permitted and the generous use of whitespace is recommended to aid legibility.

Statements and keywords are not case sensitive. The following are therefore synonymous:

```
SHOWMESSAGE “Hello, World”
ShowMessage “Hello, World”
```

In line with other languages, the use of mixed case for scripting is highly encouraged. Variable names are normally case sensitive, unless the directive $VARCASE is added to the script before any variables have been declared.

Comments can be added to a script by prefixing a line with an asterisk (*) or shriek (!). These are stripped out before compilation.

The following table lists the naming rules for the various primitive elements:

String	Strings are enclosed in single or double quotes or back slashes.
Number	Numbers are composed of digits with optional decimal points but no other formatting permitted. Presently only UK/US format decimals using a dot between the integral and fractional portions are supported.
Variables	Variables are created on use unless the $EXPLICIT flag has been set. Variable names must begin with an alpha character and may include alpha and numeric characters. IMPORTANT : Periods (.) are **not** permitted in variable names.
System Variables	System variables begin with an @ symbol. A list of these is given below.
Members	Members (properties and methods) are prefixed by a period. Property and method names are not case sensitive.
Keywords	Keywords are non case-sensitive.
Comments	Comments commence with an asterisk or shriek and are line terminated.

System Variables

The following system variables are available for mvTest:

@ACCOUNT	Account name or path
@FM	Field Mark
@DATE	Current system date
@DAY	Day number
@FALSE	False (zero)
@LOGNAME	User login name
@SM	Subvalue mark
@TIME	Current time
@TRUE	True (1)
@USERNO	Current user number
@VM	Value mark
@WHO	Account name

@TEST	Name of test
@OUT	Output buffer
@ERROR	Error buffer
@SCREEN	Screen buffer
@USER	User name
@PASSWORD	Password
@LEFT	Cursor left key
@RIGHT	Cursor right key
@UP	Cursor up key
@DOWN	Cursor down key
@HOME	Cursor home key
@END	Cursor end key
@PGUP	Page up key
@PGDN	Page down key
@F1	Function key 1
@F2	Function key 2
@F3	Function key 3
@F4	Function key 4
@F5	Function key 5
@F6	Function key 6
@F7	Function key 7
@F8	Function key 8
@F9	Function key 9
@F10	Function key 10
@F11	Function key 11
@F12	Function key 12
@EOL	End of line key

(Continues on next page ...)

@COL	Terminal column
@ROW	Terminal row
@BUFFER	Terminal buffer. Can be written to mainly to clear down the image to prevent EXPECT and WAITFOR statements seeing previous entries.
@SESSIONID	Session number
@CLIENT	True if running from the client
@PLATFORM	Platform name, currently UNIVERSE or UNIDATA.
@TRANSPORT	Transport for terminal connection: telnet, ssl or ssh
@COLS	Number of columns for the terminal.
@ROWS	Number of rows for the terminal.
@HOST	Name of the server.
@REPLY	Clean response from the server. Does not look at the current buffer, so can be used to check that the server has replied since the last send or fetch.

Standard Flow Control Constructs

Conditional Constructs

Conditional tests used for IF, CASE, UNTIL and WHILE evaluate to @TRUE if the result is a positive numeric: the Script language like UniVerse Basic interprets @TRUE as 1 and @FALSE as zero (unlike regular Windows where True is -1)

IF	The IF statement performs a simple conditional test. Only a multi-line version of the IF statement is supported.
CASE	The CASE statement provides a multiple branching conditional test:

IF syntax:

```
IF expression THEN
  statements
[END ELSE
  Statements]
END
```

CASE syntax:

```
BEGIN CASE
   CASE expression
     Statements
  [CASE expression
    Statements]
END CASE
```

Jumping Constructs

Internal subroutine jumps can be made to internal labels, which are alphanumeric and terminated by a colon, e.g. MyLabel:.

Unconditional jumps (GO, GOTO) are not supported.

GOSUB	Conditional jump GOSUB label
RETURN	Return from jump or script

Looping constructs

There are just two standard looping constructs: the counted FOR loop and the DO..REPEAT loop. Note that the NEXT keyword follows the standard modern format and is not followed by the name of counter variable.

FOR	Counted loop

```
FOR var = start TO finish [STEP step]
  Statements
NEXT
```

LOOP	Open Loop

```
LOOP
   Statements
WHILE|UNTIL condition
  Statements
REPEAT
```

Operators

The Scripting language supports the standard range of UniVerse Basic operators, including the short form assignment operators (+=, -= etc.).

Operator precedence is under review so developers should use parentheses to force precedence for safety.

+	Addition
-	Subtraction
*	Multiplication
/	Division
:	Concatenation
[n,m]	Substring extraction operator
[d,n,m]	Field operator
<f,v,s>	Dynamic array operator

Comparison Operators

= or EQ	Equality
> Or GT	Greater Than
>= or GE	Greater or Equal to
< or LT	Less than
<= or LE	Less or Equal to
<> # or NE	Not Equal
AND	And
OR	Or

Short form operators:

+=	Additive
-=	Subtractive
/=	Divisive
*=	Multiplicative
:=	Concatenative

Accessing Properties and Methods

One major difference between Universe Basic and the scripting language is the support for calling properties and methods of tools and static objects. This is currently limited to the client runner, though limited support will be added to the server scripting engine at a later date.

Properties are attributes of an object that can be retrieved or modified by name. For example, to change the TITLE of a tool you can access its TITLE property as follows:

```
TEST.Title = “New Title”
```

The MyTool above is a tool handle, or a reference to the tool.

Methods are functions that are exposed by an addressable object. These are called using the syntax:

```
Object.Method
```

Or

```
Object.Method(args)
```

For example, a CSV Data Source supports a Next method that populates a variable:

```
myDataSource.Next(variable)
```

Static Objects

Static objects are part of the environment and are referred to by name. These provide information and runtime functionality about the application and the Windows client, and are detailed in the section on Static Objects below.

```
TestData = CLIENT.FileRead(“C:\temp\testdata.dat”)
```

Standard Script Statements

The Scripting Language statements include a cut down list of the most commonly used UniVerse statements that are sensible for client/server and similar operations and additional statements specific to the host environment.

When used from a client hosting language any Universe operations that are not in this list or that require close interaction with the server – for example, handling sequential files for client scripting - should be devolved to Basic subroutines and called from the scripting language. In most cases the client host will provide additional features for manipulating client side files using static objects.

Those statements that are marked with an asterisk (*) are not permitted from client scripts.

ABORT	Exit script
EXEC scriptname (args)	Execute a script.
CALL subroutine (args)	Call a subroutine on the server
CLEARDATA	Clear data stack
CLEARFILE filevar	Clear the content of the data file opened to filevar.
CLEARSELECT [ALL]	Clear the active select list*
CLS	Clear terminal window
CONVERT old TO new IN variable	Character conversions.
CREATE filevar	Create a new sequential file opened to filevar*
CRT text	Output text onto terminal window
DATA expression	Add expression to the data stack*
DEBUG	Enter script debugger
DEL variable<spec>	Delete dynamic array element
DELETE file, item	Delete item from a file
DELETELIST listname	Delete a saved select list
DISPLAY text	Synonym for CRT
EXECUTE statement [CAPTURING text]	Execute command on the server
FIND expression IN variable SETTING fno,vno,svno {THEN|ELSE}	Search a dynamic array for a given field, value or subvalue and set the levels.
FINDSTR expression IN variable SETTING fno,vno,svno {THEN|ELSE}	Search a dynamic array for a given field, value or subvalue containing expression and set the levels.
GETLIST name	Get a saved select list
INPUT variable	Input a value into a variable (shows input box on client)
INS value BEFORE variable<spec>	Insert dynamic array element
LOCATE value IN array [BY mask] SETTING pos {THEN|ELSE	Locate element in dynamic array
NULL	Null operation
OPEN file TO FV {THEN|ELSE}	Open file on server
OPENPATH path To FV {THEN|ELSE}	Open a file by path name*
OPENSEQ path To FV {THEN|ELSE{	Open a file for sequential access*
PERFORM statement	Perform a command on the server
PRINT expression	Render expression to the printer*

PRINTER ON|OFF|CLOSE	Control printer access*
RANDOMIZE	Seed the random number generator*
READ var FROM FV, Item {THEN|ELSE}	Read a record from the server
READBLK var FROM FV, length {THEN|ELSE}	Read a block from a sequential file*
READLIST var {THEN|ELSE}	Read a list from the server
READSEQ var FROM FV {THEN|ELSE}	Read a line from a sequential file*
READU var FROM FV, item {THEN|ELSE}	Read and lock item on server
RELEASE FV, Id	Release record lock
SEEK fv, offset, relto	Seek within a sequential file*
SELECT fv	Select against a file variable*
SSELECT fv	Sort-Select against a file variable*
SHOWERROR text	Show an error message
SHOWMESSAGE text	Show a message
STOP	Stop script
TRACE text	Send text to the trace window if showing.
WEOFSEQ fv	Write end of file to a sequential file*
WRITE var ON FV, Item	Write record to the server
WRITEBLK var ON fv {THEN|ELSE}	Write a block to a sequential file*
WRITESEQ var ON fv {THEN|ELSE}	Write a line to a sequential file*
WRITEU var ON FV, Item	Write record to the server preserving lock

Standard Script Functions:

The script language includes a set of intrinsic functions that broadly map to the equivalent functions in UniVerse Basic, along with some that provide specific information and interaction.

As with the standard statements, these are complemented with additional functions specific to the host environment.

ABS(value)	Return absolute value
ALPHA(value)	Return true if value is alpha
AVG(array)	Return average of a dynamic arrray
CHANGE(value, old, new)	Return value with all old changed to new
CHAR(value)	Return ascii character value
COL1()	Return start of the last FIELD() extraction.
COL2()	Return end of the last FIELD() extraction.
CONVERT(old, new, value)	Return value with old chars converted to new
COUNT(value, delim)	Return count of delim in value
DATE()	Return current date (Universe format)
DCOUNT(value, delim)	Return count of delimited entries in value
DELETE(array, fno, vno, svno)	Return array with element deleted
DOWNCASE(value)	Return value converted to lower case
DQUOTE(value)	Return value enclosed in double quotes
EXTRACT(array, fno, vno, svno)	Return element extracted from array
FIELD(value, delim, start, take)	Return delimited substrings from value
FIX(value, dp)	Return value fixed to a number of decimal places
FOLD(value, width)	Return value with field marks inserted at every width or preceding space to fit into a region.
ICONV(value, code)	Return value converted to internal format
INDEX(value, substring, occur)	Return position of occurrence of substring
INSERT(array, fno, vno, svno, value)	Return array with element inserted
INT(value)	Return integer portion of value
LEFT(value,n)	Return leftmost characters from value
LEN(value)	Return length of value
LOWER(value)	Return value with delimiters lowered
MAXIMUM(array)	Return maximum value from array
MINIMUM(array)	Return minimum value from array
MOD(value, divisor)	Return modulus
MID(value, n, m)	Return substring from middle of value
NOT(value)	Return logical negation of value
NUM(value)	Return true if value is a number
OCONV(value, code)	Return output conversion of value
PWR(value, n)	Return value raised to the power n
RAISE(value)	Return value with delimiters raised
REPLACE(array, fno, vno, sno, value)	Return array with element replaced
RIGHT(value, n)	Return rightmost characters from value

SEQ(value)	Return ASCII number for a character.
SOUNDEX(value)	Return soundex encoding for value.
SPACE(n)	Return string of n spaces
STR(value, n)	Return string of n occurrences of value
SUM(array)	Return sum of array elements
TIME()	Return current time Universe format
TIMEDATE()	Return date and time in external format.
TRIM(value)	Return value less extraneous spaces
TRIMB(value)	Return value less trailing spaces
TRIMF(value)	Return value less leading spaces
UPCASE(value)	Return value converted to upper case

Using EQUATEd Literals

The script language, just like Basic, has a pre-compiler step that resolves tokens to literal values.

This is based on the same EQUATE syntax as used by Basic, making it feasible to use the same equate names and even the same include files for both your scripting and back end code.

The following lists the compiler directives supported by the script compiler:

$INCLUDE filename itemname	Includes external source source. This is typically used for files of EQUATEd values.
$VARCASE	If this directive appears, variable names are compiled to be non-case-sensitive.
$EXPLICIT	if this directive appears, variables must be declared before use using a DIM statement.
$DESCRIPTION	This sets the description into the repository header.

HINT

Use include files for your login scripts.

Using Session Variables

Session variables are defined using a COMMON or SESSION declaration. However these operate differently than the way COMMON is handled in Basic.

Session variables in the scripting language are owned by the script runner and persist for the lifetime of the connection. You can declare session variables as follows:

```
SESSION MYSESSIONVAR, MYSESSIONVAR2 …
```

Session variables, unlike their Basic equivalent, are stored and referenced by name and not by their position in a common block. Unlike named common, session variables are initially set to an empty string and not to zero.

Standard Scripting Statements

ABORT

The ABORT statement terminates a script.

Unlike the regular STOP statement, the ABORT statement signals to the test runner that no feedback on this script should be presented.

This provides a way for closing a script without the results being logged, for example, where the running of the script is conditional on a status variable or test run variable.

Example:

```
* How many sessions should run this particular test?
MaxSessions = get("MAX_SESSIONS")
IF @SESSIONID >= MaxSessions Then
   ABORT
END
```

See also:

STOP

ABS()

The ABS function returns the absolute value of an expression.

```
If QTY < 0 Then
   QTY = ABS(QTY)
End
```

ALPHA()

The ALPHA function returns TRUE if an expression consists entirely of alpha characters.

```
ALPHA(expression)
```

Example

```
* Wait for status code to appear
WaitUntil (ALPHA(SNAG(5,5,10)))
```

AVG()

The AVG function returns the average of a set of numbers passed as a dynamic array.

```
AVG(Array)
```

Example

```
AverageTime = AVG( ALLTIMES )
```

CALL subroutine

The CALL Statement calls a BASIC subroutine on the server.

```
CALL Subroutine( args )
CALL @subroutine(args)
```

The CALL Statement is restricted to 19 arguments on the server side script runner.
For client side scripts, only arguments that can be represented as strings can be passed: that includes numbers, strings, dynamic arrays but not file variables, dimensioned arrays, select variables or handles.

This restriction is imposed by the middleware used.

CHANGE()

The CHANGE() function returns a copy of an expression with all instances of a substring substituted with a new substring.

Result = CHANGE(expression, old, new)

Example:

```
Result = Change( Line, ",", @FM)
```

See also:

CONVERT

CHAR()

The CHAR function returns the character with the specified ASCII value.

```
ESC = CHAR(27)
```

Note that SEND expressions can include special characters that are automatically substituted. See SEND for details.

CLEARDATA

The CLEARDATA statement clears the data stack.

CLEARDATA

This is ignored by the client script runner.

CLEARFILE fv

The CLEARFILE statement clears the content of a database file. The file must have been previously opened to a file variable using the OPEN statement.

```
Open 'MYTESTDATA' To TESTFL Then
  CLEARFILE TESTFL
End
```

CLEARSELECT

The CLEARSELECT statement clears the currently active select list.

```
CLEARSELECT
```

This statement is ignored by the client script runner. If you need this from the client you will have to EXECUTE the CLEARSELECT TCL command.

COL1()

The COL1() function returns the starting location of the last FIELD() function extraction.

COL2()

The COL2() function returns the ending location of the last FIELD() function extraction.

CONVERT, CONVERT()

The CONVERT statement modifies the content of a variable by replacing all characters in one list with the equivalent characters in a second list.

```
CONVERT old TO new IN variable
```

Example:

```
CONVERT “,” TO @FM IN TestData
```

The CONVERT() function does the same in function format:

```
Result = Convert(“,”,@FM, TestData)
```

See also:

CHANGE() function

COUNT()

The COUNT function counts the number of occurrences of a substring in an expression.

Example:

```
If Count(AllowedCodes, Code) Then
```

CREATE filevar

The CREATE statement creates a new sequential file.

The file must have been opened to the filevar using the OPENSEQ or OPENBLK statements before calling CREATE.

This statement is not supported by the client side script runner.

Example:

```
Open ‘log.dat’ To LOGFILE Else
  Create LOGFILE Else
    Crt ‘Cannot create log file’
   STOP
 End
End
```

CRT expression

The CRT statement updates the output text for the test with an expression.

Example:

```
CRT “Sending user name : “
Send @USER
```

DATA Expression

The DATA statement appends an expression to the DATA stack.

```
DATA “YES”
```

This is ignored by the client side test runner.

DATE()

The DATE() function returns the current date in INTERNAL format.

DCOUNT()

The DCOUNT() function returns the number of substring separated by a delimiter in an expression.

```
DCOUNT( expression,  delim)
```

Example:

```
NumLines = DCOUNT( Lines, @FM )
```

DEBUG

This enters the test script debugger.

DEL variable < spec >

The DEL statement deletes a specified field, value or subvalue from a dynamic array.

```
First = MyData<1>
Del MyData<1>
```

DELETE filevar, key

The DELETE statement deletes a record from a database file.

The file must have been opened to the file variable.

```
Open 'MYRESULTS' To F_RESULTS Then
  Delete F_RESULTS, 'CURRENT_RESULTS'
End
```

DELETELIST listName

The DELETELIST statement deletes a previously saved select list.

DOWNCASE() DCASE()

The DOWNCASE or DCASE function returns a string converted to lower case.

```
TestAns = DownCase( Answer )
```

DQUOTE()

The DQUOTE function returns an expression wrapped in double quotation marks.

```
Command = “LIST CUSTOMER WITH SURNAME = “ : DQUOTE(Surname)
```

EXECUTE statement

The EXECUTE statement runs a command on the server.

```
EXECUTE statement [CAPTURING output]
```

The text of the command may optionally be captured to a variable.

Example:

```
EXECUTE “LIST.READU EVERY” CAPTURING LOCKLIST
```

FIELD()

The FIELD() function extracts one or more delimited substrings from an expression. You can also use the field operator.

```
FIELD( expression, delimiter, start, take )
```

Example:

```
SecondWord = Field( Words, “ “, 2, 1)

RestOfLine = Words[“ “,2, Len(Words)]
```

FOLD()

The FOLD function returns text folded to a specified width. The text will word wrap to that width is possible.

Result = FOLD(expression, width)

ICONV()

The ICONV() function performs an input conversion on a value.

Result = ICONV(expression, code)

Note that conversions take place on the server. Standard date and time conversions should therefore be avoided when running scripts on the client: see the CLIENT object methods.

INDEX()

The INDEX() function returns the character position of an occurrence of a substring in an expression, starting from 1. If the substring is not found zero is returned.

Result = INDEX(expression, substring, occurrence)

Example:

```
If INDEX( Answers, Answer, 1 ) = 0 Then
   Error “Not a valid answer”
   STOP
End
```

INPUT variable

The INPUT statement enters a user supplied value into a variable.

INPUT variable

The user is prompted with an input box by the client side script runner, or a regular input by the server runner.

This of course means that the script cannot be fully automated. Where possible use the GET() function to retrieve run data instead (see TEST Scripting Statements below).

INS

The INS statement inserts an element in front on an existing field, value, or subvalue in a dynamic array.

Example:

```
INS Date() Before List<1>
```

INT()

The INT() function returns the integer portion of a floating point expression.

```
Result = INT( Expression )
```

LEFT()

The LEFT() function returns the leftmost characters of an expression.

```
Result  = LEFT( expression, number )
```

Example:

```
Initial = LEFT( Forename, 1)

Initial = Forename[1,1]
```

LEN()

The LEN() function returns the length of an expression.

```
Result = LEN( expression  )
```

Example

```
If LEN(TRIM( SNAG( 10, 10, 5)) > 0 Then
```

LOCATE

The LOCATE statement searches a dynamic array for a given field and returns the corresponding field number.

```
LOCATE value IN array [BY mask] SETTING pos {THEN|ELSE}
```

This follows the PICK format of LOCATE.

Example:

```
Locate Value In AllowedValues Setting Pos Else
  Error ‘Value is not allowed’
  STOP
End
```

LOWER()

The LOWER() function returns an array in which all delimiters have been lowered.

```
Result  = LOWER( Expression )
```

Example:

```
Details< LastLine > = LOWER( Record )
```

See also:

RAISE()

MAXIMUM() and MINIMUM()

The MAXIMUM and MINIMUM functions return the maximum and minimum values respectively from a set of numbers passed as a dynamic array.

```
Result = MAXIMUM( array )
Result = MINIMUM( array )
```

MOD

The MOD() function returns the modulus (remainder) of two expressions.

```
Result = MOD( expression, divisor )
```

NULL

The NULL statement does nothing.

```
Open ‘MYFILE’ TO FL Else
   NULL
End
```

OCONV()

The OCONV() function performs an output conversion on a value.

```
Result = OCONV( expression, code )
```

Note that conversions take place on the server. Standard date and time conversions should therefore be avoided when running scripts on the client: see the CLIENT object methods.

OPEN

The OPEN statement opens a database file and associates it with a file variable. The file variable is passed to delete, read and write statements.

```
Open ‘MYFILE’ TO FL Else
   NULL
End
```

OPENPATH

The OPENPATH statement opens a database file by its underlying path name and associates it with a file variable. This is only supported on UniVerse and only for server side scripts.

```
OPENPATH ‘../mylogs/LOGDATA’ To FL Else
```

OPENSEQ

The OPENSEQ command opens a file for sequential operations.

```
OPENSEQ ‘c:\logs\data.log’ TO LOGFL Else
   Crt ‘Cannot open log file’
   STOP
End
```

This is only supported by the server side script runner. Client side files can be manipulated using the CLIENT static object methods.

PERFORM statement

The PERFORM statement runs a TCL command.

```
PERFORM “CLEARCOMMON ALL”
```

PRINT

The PRINT statement sends an expression to the current print job. This should be followed by a PRINTER ON command.

```
PRINT Results
```

The PRINT statement is not supported by the client side script runner.

PRINTER ON|OFF|CLOSE

The PRINTER statement turns on or off routing to the system printer.

This is only supported by the server side script runner.

PWR()

The PWR function returns a number raised to the n-th power.

```
Result = PWR( number, n)
```

RAISE()

The RAISE function returns a dynamic array with all delimiters raised.

```
Result  = RAISE( Expression )
```

Example:

```
TestData = RAISE( DetailLine<Counter> )
```

RANDOMIZE

The RANDOMIZE command seeds the random number generator used by the RND function.

Note that this does not affect the ONE() test function that uses its own randomizer.

READ

The READ statement reads a record from a file. The file must have been previously opened to a file variable.

```
READ record FROM filevar, key {THEN|ELSE}
```

Example:

```
READ OrderRec From ORDERFL, NewOrder Else
  Error ‘New order has not been written’
  STOP
End
```

READBLK

The READBLK statement reads a block of data of up to a specified length from a sequential file and assigns it to variable.

```
READBLK variable FROM fv, length
```

This is only supported by the server side script runner.
Client files can be accessed using the CLIENT static object methods.

READSEQ

The READSEQ statement reads a line of data from a sequential file and assigns it to variable.

```
READSEQ variable FROM fv, length
```

This is only supported by the server side script runner.
Client files can be accessed using the CLIENT static object methods.

READU

The READU statement locks and reads a record from a file. The file must have been previously opened to a file variable.

```
READU record FROM filevar, key {LOCKED|THEN|ELSE}
```

Example:

```
READU OrderRec From ORDERFL, NewOrder Else
   Error 'New order has not been written'
   STOP
End
```

RECORDLOCKED()

The RECORDLOCKED() function returns a Boolean stating whether the specified record currently has an item lock present.

```
Result = RecordLocked(filevar, key)
```

RELEASE

The RELEASE statement releases a record lock.

```
RELEASE filevar, key
```

Only this form of the release statement is supported. This is by design.

RIGHT()

The RIGHT function returns the rightmost characters from an expression.

Result = RIGHT(expression, number)

Example:

EndOFString = Right(value, 1)

SEEK

The SEEK statement positions the read/write point in a sequential file.

SEEK fv, offset, relto

SEEK is only supported by the server side script runner.

SEQ

The SEQ() function returns the ASCII sequence number of the initial character of an expression.

Result = SEQ(expression)

SHOWERROR

The SHOWERROR statement logs an error.

SHOWERROR “Cannot connect to the database”

Note that validation errors should use ASSERT handling (below).

SPACE()

The SPACE() function returns a string of spaces.

Result = SPACE(number)

STR()

The STR() function returns a string repeated a number of times.

Result = STR(expression, number)

Example;

Crt Str(“-“,78)

STOP

The STOP statement unconditionally stops a running script. This returns the script status to the test runner.

STOP

See also:

ABORT

SUM()

The SUM function returns the total of a set of numbers passed as a dynamic array.

Result = SUM(array)

Example

Balance = SUM(OrderQuantities)

TIME()

The TIME() function return the time in INTERNAL format.

Result = TIME()

TIMEDATE()

The TIMEDATE() function returns the time and date in an EXTERNAL format.

Result = TIMEDATE()

NOTE: this should be used in place of OCONV(@DATE) or OCONV(@TIME) to prevent round trips to the server.

TRIM(), TRIMB(), TRIMF()

The TRIM(), TRIMB() and TRIMF() function strip extraneous, trailing or leading whitespace from an expression.

Result = TRIM(expression)

UPCASE()

The UPCASE() function returns an expression converted to upper case.

Result = UPCASE(Expression)

WEOFSEQ

The WEOFSEQ statement writes and end of file mark to a sequential file.

WEOFSEQ filevar

This is only supported by the server side script runner.

WRITE

The WRITE statement writes a record onto a database file.

The file must have been previously opened to a file variable.

```
OPEN ‘RESULTS’ TO F_RESULTS Then
   WRITE ResultData On F_RESULTS, @DATE
END
```

WRITEBLK

The WRITEBLK statement writes a block of data onto a file opened for sequential access.

WRITEBLK variable ON FV {THEN|ELSE}

This is only supported by the server side script runner.
Client side files can be manipulated through the CLIENT static object.

WRITESEQ

The WRITESEQ statement writes a line of data onto a file opened for sequential access.

WRITESEQ variable ON FV {THEN|ELSE}

This is only supported by the server side script runner.
Client side files can be manipulated through the CLIENT static object.

CHAPTER 4 Test Script Statements

This section lists the additional statements and functions specific to the test scripts.

ANNOUNCE "text"

The ANNOUNCE statement introduces a test.

The text supplied to the announcement will be automatically added to the output for each assertion in that test, to prevent the need to duplicate some standard informative text for each assertion message.

```
ANNOUNCE "Check account balance is in credit"
```

The announcement text will remain in force until the end of the test, or until a new ANNOUNCE statement is encountered. This allows it to operate across internal subroutine calls.

CHECK(value, code)

The CHECK() function checks a value for consistency using one of a number of coded validations. These are performed locally and are quicker than the equivalent coding for each validation in turn, and may be externally defined.

The check code contains one or more check conditions.
Each check condition is separated from its neighbour by a semicolon.

The condition is preceded by a single character check type and a colon, as follows:

F:filename

The check conditions are:

C	code	Checks value against a conversion code
F	Filename	Checks that the value is a key to a record on filename
L	length	Checks the length of value
P	Pattern	Checks value against a pattern match
R	low-high	Checks value against a range
T	N,I,T,D	Checks value is a number, integer, date or time
V	list	checks value against a comma separated list

Example:

CHECK(value, “T:N;F:PARTS”)

CLEARCOMMON

The CLEARCOMMON statement clears named common blocks on the server where supported by the database (not available on UniData).

Executing a regular CLEARCOMMON ALL will wipe the named common block used by mvTest, with undefined consequences. The CLEARCOMMON script statement temporarily stores the mvTest common, issues a CLEARCOMMON ALL and then restores the mvTest common safely.

This is only supported by the client side runner. Attempts to clear common from the server will be blocked by the database itself.

The use of CLEARCOMMON is NOT recommended.

Example:

```
* Clear all named common blocks

CLEARCOMMON
```

CONNECT

The CONNECT statement opens a new TELNET connection to your server. For reasons of security, this uses the same host name details as the test client connection if you are running through the test client, and localhost if you are running using the server test rig.

Example:

```
* Connect a new session

CONNECT
```

See also:

DISCONNECT

CURSORAT(x, y)

The CURSORAT() function returns true if the cursor has settled at the specified x, y location.

The @COL and @ROW variables hold the current location of the cursor at the moment they are requested. The CURSORAT() and CURSORPOS() functions test the cursor at 100ms intervals until it settles at a location. This is therefore a more accurate test when checking for input locations.

Example:

```
WaitUntil (CursorAt( 20, 10))
```

See also:

CURSORPOS

CURSORPOS()

The CURSORPOS() function returns the settled current location of the cursor.

The @COL and @ROW variables hold the current location of the cursor at the moment they are requested. The CURSORAT() and CURSORPOS() functions test the cursor at 100ms intervals until it settles at a location. This is therefore a more accurate test when checking for input locations.

Example:

```
Position = CursorPos()
X = Position<1>
Y = Position<2>
```

See also:

CURSORAT

DISCONNECT

The DISCONNECT statement disconnects an active TELNET connection.

Example:

```
DISCONNECT
```

See also:

CONNECT

ERROR text

The ERROR statement raises an error condition with the specified text.

Example:

```
EXPECT SomeText ELSE
   Error “Could not find “:SomeText
   STOP
END
```

EXPECT text {THEN|ELSE}

The EXPECT statement wait for up to MAXINTERVAL seconds for the specified text to appear in the input buffer. If the text does not appear in time, the ELSE clause will be fired.

Example:

```
Expect “Record Saved” Else

   STOP
   DISCONNECT
End
```

See also:

LOOKFOR
WAITFOR
WAITUNTIL
WANT

FETCH

The FETCH statement polls the terminal for any new data received.
This can be used to bring the buffer up to date before accessing @SCREEN or @BUFFER.

Example:

```
FETCH
If Index(@SCREEN, Text, 1) Then
  …
End
```

GET(key)

The GET function returns a variable stored on the active test run definition.

This is primarily to allow test runs to be customized, for example as part of a support environment for different customer sites.

Example:

```
ThisCustomer = GET(“CustomerName”)
```

LOOKFOR(dynamic_array, timeout)

The LOOKFOR() function looks in the buffer for one of a number of possible strings supplied as a dynamic array. If one of the strings is found, the index of that string (one based) is returned. If none of the strings have been found within the timeout period, the function returns @FALSE.

This is useful for scripts that need to anticipate possible error messages raised by the application when running with test data. Using LookFor you can check for one of these messages appearing in response to a data entry.

Example:

```
ErrorMessages = “Price Error” :@FM: “Mandatory Field”
Found = LookFor( ErrorMessages, 5)

If Found Then
  * handle errors
End
```

See Also:

EXPECT
WAITFOR
WAITUNTIL
WANT

NAP milliseconds

The NAP statement waits unconditionally for the specified number of milliseconds before continuing.

Example:

```
SEND Name
NAP 100
```

See also:

WAIT

ONE(specification)

The ONE function randomly returns one of a set of values. This allows you to create a pool of values for generating test data and conditions.

The specification determines from where the ONE() function will retrieve its candidate list of values. This can be one of the following:

An @FM delimited series of fields.
A comma delimited series of values.
A range of values in the format LOW-HIGH.
A data file name.

If a data file is specified, the ONE function will retrieve a random sample of 100 record keys the first time it is called with that file name. This will be stored for the duration of the test run.

Example:

```
TestQty = ONE(1-10)
```

PASSWORD

The PASSWORD statement sends the password used for the test client connection when running from the test client, or the stored password when running from the server test rig.

It is equivalent to SEND @PASSWORD and included for backward compatibility.

Example:

```
WAITFOR “word:”
PASSWORD
```

See also:

SEND
@PASSWORD

PUT expression

The PUT statement sends data to the server through the active TELNET connection.
Unlike SEND, the PUT statement does not add a carriage return to the end of the expression.

PUT should be used when sending special characters such as the escape key or function keys, or where the application responds to a single character entry.

Example:

```
* Send F2 to save the screen
PUT @F2
```

See also:

SEND

READ(filename, key)

The READ function performs a quick read on a data file given a specified key and returns the record body. If the record does not exist an empty string is returned.

Example:

```
Rec = Read("ORDERS", OrderNumber)
AssertFull "Record should not be empty", Rec
```

See also:

OPEN statement
READ statement

SEND Expression

The SEND statement sends the results of expression to the server through the TELNET connection as regular terminal input. SEND adds a carriage return to the end of the text to simulate a user pressing return on an input.

The expression may contain the following substitution characters:

\\	single backslash
\r	Carriage return
\n	Line feed
\b	Backspace
\t	Tab character
\e	Escape character

Example:

Send “SMITH”

See also:

PUT

SNAG(x, y, length)

The SNAG function returns the textual content of the terminal screen at the cursor location x, y and continuing for length characters.

Example:

```
Test = SNAG( 0, 10, 20 )
```

STRIP(value, code)

The STRIP function returns a substring of characters stripped from value based on the character types. This is most useful in combination with the SNAG() function to remove background or mask characters from a snagged area.

This is more efficient in client run scripts than performing the equivalent OCONV functions that require a round trip to the server through the shared control connection.

The codes are:

/	Inverts the entire mask
N	numeric characters
A	alpha characters
D	numeric characters and decimal point

For example:

```
OrderNumber = Strip(Snag(0,20,20),"N")
```

WAIT seconds

The WAIT statement pauses unconditionally for a specified number of seconds before continuing.

Example:

```
WAIT 10
```

See also:

NAP

WAITFOR text

The WAITFOR statement waits for up to MAXINTERVAL seconds for the specified text to appear in the buffer. If the text is not found within the given time, the script will terminate with a FAIL condition.

Example:

```
WAITFOR “Password:”
```

See also:

EXPECT
LOOKFOR
WAITUNTIL
WANT

WAITUNTIL condition

The WAITUNTIL statement waits for up to MAXINTERVAL for a condition to become true. This provides the most powerful of the waiting options, but can be the most difficult to code as only a single condition is allowed.

An example might be to look for an error message or the signal that you can continue an entry, as below.

Example:

```
ErrorMessages = “Mandatory Field” : @FM : “Price error”
WaitUntil ( LookFor(ErrorMessages,1) OR (@COL = 10 AND @ROW = 12))
```

See also:

EXPECT
LOOKFOR
WAITFOR
WANT

WANT x, y, text,timeout {THEN|ELSE}

The WANT statement waits for up to timeout seconds for the specified text to appear at a specific position on the screen.

If the text does not appear within the timeout the ELSE clause is fired.

Example:

```
WANT 0,22,”Error : “,1 THEN
   * handle the error
END
```

See Also:

EXPECT
LOOKFOR
WAITFOR
WAITUNTIL

Assertions and Test Conditions

Assertions sit at the heart of structured testing.

An assertion is a check on the status of the test. The results of the assertion are logged against the test, and the test status is set to fail if the assertion fails.

Each assertion consists of a message and a test condition. The message appears in the output pane of the Test Result window and if the test fails, this sets the status message on the test.

The simple form of an assertion is:

Assert message test

For example:

Assert “Balance must be greater than zero”, (Balance > 0)

The other assertion options are variations on the theme to simplify and clarify the scripts.

ASSERT message, condition

The ASSERT statement evaluates condition and returns a pass or fail status with the selected message.

Example:

ASSERT “Balance should be greater than zero”, Balance > 0

ASSERTBETWEEN message, low, high, actual

The ASSERTBETWEEN test checks that a test value is within a certain range. You could also use a CHECK() with a range specification, but this is clearer.

Example:

ASSERTBETWEEN "Should be within sensible range", 1, 100, Qty

ASSERTIS message, test1, test2

The ASSERTIS statement evaluates two test values and returns a pass status if the two values match.

Example:

ASSERTIS “Date should be today”, SomeDate, @DATE

ASSERTNOT message, test1, test2

The ASSERTNOT statement evaluates two test values and returns a pass status if the two values do not match.

Example

ASSERTNOT “Value should not be empty”, Value, “”

ASSERTCONT message, value, substring

The ASSERTCONT statement evaluates a test value and returns a pass status if the test value contains a specified substring.

Example

ASSERTCONT “Value should include a period”, Value, “.”

ASSERTMATCH message, pattern, value

The ASSERTMATCH statement evaluates a test value and returns a pass status if the test value matches a Basic pattern match.

Example

ASSERTMATCH “Value should be a valid sort code, “2N’-‘2N’-‘2N”, Value

ASSERTEMPTY message, value

The ASSERTEMPTY statement evaluates a test value and returns a pass status if the value is an empty string.

Example:

ASSERTEMPTY “Error should be empty”, Error

ASSERTEXISTS message, fileName, itemName

The ASSERTEXISTS statement checks whether an item exists on a file.

Example:

ASSERTEXISTS “The invoice should have been written”, “INVOICES”, InvoiceId

ASSERTFULL message, value

The ASSERTFULL statement evaluates a test value and returns a pass status if the value is not an empty string.

You can use ASSERTNOTEMPTY as a synonym for ASSERTFULL.

Example:

```
ASSERTFULL “Line total should not be empty”, LineTotal
ASSERTNOTEMPTY "There should be no error", ErrText
```

ASSERTHAS message, test, value

The ASSERTHAS statement evaluates a test value and returns a pass status if the value includes a test value as a dynamic array element.

Example:

```
ASSERTHAS “Order lines should include A123”, “A123”, OrderLines
```

ASSERTHASNOT message, test, value

The ASSERTHASNT statement evaluates a test value and returns a pass status if the value does not include a test value as a dynamic array element.

Example:

```
ASSERTHASNT “Order lines should not include A123”, “A123”, OrderLines
```

ASSERTLOCKED message, fileName, itemName

The ASSERTLOCKED and ASSERTNOTLOCKED statements check whether an item lock has been set on a record. This will test both for locks set by the current session and for locks for other users, to permit its use in UI testing.

Example:

```
ASSERTLOCKED “Invoice should be locked”, “INVOICES”, InvoiceId
ASSERTNOTLOCKED "Order should be released", "ORDERS", OrderId
```

ASSERTWITHIN message, expected, delta, actual

The ASSERTWITHIN test checks that a test value is within a certain delta (positive or negative) of an expected value. This is most useful for tests against the current time, where the clock may have moved forward a second or so between the production of the result and the test taking place.

Example:

```
CALL SomeLongRoutine(InData,OutData,ErrText)
* Should have todays time (within 5 seconds or so)
TimeReturned  = OutData<1>
ASSERTWITHIN "Should be about now", Time(), 5, TimeReturned
```

STATIC OBJECTS

The Script Language exposes the following static objects:

CLIENT OBJECT

The CLIENT object provides members for integrating scripts with the runtime environment, especially when running through the client side script runner. Most of these are also handled through the server side runner, providing a better (more transportable) mechanism for handling the file system.

New properties and methods may be added so please check the online documentation for the most up to date list of features.

Example:

```
ShouldFileAdd:
  CLIENT.FileAdd('TESTDIR\NEWFILE.txt','FIRST LINE')
  CLIENT.FileAdd( 'TESTDIR\NEWFILE.txt','SECOND LINE')

  NewRec3 = Read("TESTDIR", "NEWFILE.txt")

  AssertCont "fileadd should have first line", NewRec3, "FIRST LINE"
  AssertCont "fileadd should have second line", NewRec3, "SECOND LINE"

Return
```

Client Properties

CLIPBOARD	Gets or sets the clipboard text.*
CURRENTDIRECTORY	Gets or sets the current directory.
ERROR	Returns the last Windows error.*
SCREENCOUNT	Returns the number of physical screens attached*
SCREENWIDTH	Returns the primary screen width in pixels.*
SCREENHEIGHT	Returns the primary screen height in pixels.*
DRIVES	Returns a list of the drives attached to the PC.*
OSTYPE	Returns 'WINDOWS' or 'UNIX'
CLIENTTYPE	Returns 1 for server runner or 0 for the client runner.

* client runner only.

File System Methods

AVAIL(result, drive)
result = AVAIL(drive)
	Returns the available space on the drive in 1KB blocks.*

DIR(result, path)
Result = DIR(path)
	Returns a list of all files in path.

MKDIR(path)	Create local directory

DIREXISTS(result, path)
Result= DIREXISTS(path)
	Check whether a local directory exists

FILEADD(path, data)	Append data to a local file with line breaks.
FILEDELETE(path)	Delete a local file
FILECOPY(path, newpath)	Copy a local file

FILEDATE(result, path)
Result = FILEDATE(path)
	Returns a file update date as a UniVerse format date.

FILEEXISTS(result, path)
Result = FILEEXISTS(path)
	Checks whether a local file exists

FILETIME(result, path)
Result = FILETIME(path)
	Returns a file update time as a UniVerse format time.

FILEREAD(result, path)
Result = FILEREAD(path)
	Read data from a file

FILEREADBLK(result,path,offset,size)
Result = FILEREADBLK(path,offset,size)
	Reads up to size bytes from file starting at offset.

FILERENAME(path, newpath)	Rename a local file

FILESIZE(result,path)
Result = FILESIZE(path)
	Gets the size of a file.

FILEWRITE(path, data)	Overwrite a local file with data
RMDIR(path, force)	Removes a directory, use force if not empty.

Process Handling

EXECUTE(command, arguments)	Executes a Windows or operating system command and waits until complete
NAP(interval)	Pauses the current thread for interval milliseconds (UniVerse or client runner only)
RUN(Command, arguments)	Executes a Windows command and returns without waiting*

RUNNING(result, command)
Result = Running(command)
	Checks whether a command is running*

SHELLOPEN(Document)	Opens a document using the Windows shell, by opening the application associated with the document type.*
SLEEP(interval)	Pauses the current thread for interval seconds.

* client only

Showing Dialogs

All these are client runner only.

SHOWOPEN(title, filter, filename)	Runs a standard Windows File Open dialog to request a file name. If cancelled the Filename is returned as an empty string.*
SHOWSAVE(title, filter, filename)	Runs a standard Windows File Save dialog to request a file name. If cancelled the Filename is returned as an empty string.*
SHOWPRINTER	Runs a standard Windows printer setup dialog.*

RUNNER OBJECT

The RUNNER Object represents the script runner.

LoadCSV(name, source, handle)	Loads a CSV Data Source and returns a handle to the source. When called from the client runner the source is given a name that ensures it will only be loaded once for all running sessions. The name is ignored by when called from the server runner as it is not possible to share resources between sessions.

Data Source Objects

The Data Source Objects are loaded by the RUNNER static object and define specific custom data sources. They have the following methods:

INIT	Initialize the data source
NEXT(value)	return the next value from the source or an empty string.
CYCLE(value)	return the next value from the source or return to the start
SEED(value)	Move the current point to value
CLOSE	Close and clear the source
COLS	Get a list of columns
GET(name)	Get the named column from the current row
SHUFFLE()	Shuffle the data into a random order

LOCK Object

The LOCK object provides an abstraction to the lock phantom used to assist multi-user testing simulations. This is described later in this User Guide.

Clear()	This clears down the counters and arguments captured for a mock without tearing down the mock itself.
Called()	This returns the number of times the mock routine has been called.
Check(arg,specification)	This performs an mvTest Check() function on the value passed into the requested argument.
Restore()	This tears down the mock and restores the original routine.
Want(arg)	This sets the desired return value of a given argument.
Was(arg)	This returns the value that was sent for a given argument.
WasCalled()	Returns true if the routine was called at least once.
WasNotCalled()	Returns true if the routine was never called.

Utility Commands

The following utility commands have been added:

TEST.COPY	Copy a test to a new name.
TEST.DELETE	Delete a test and its associated script.
TEST.RENAME	Rename a test and its associated script.
TEST.UNDELETE	Restore a previously deleted test.
TEST.CHECKOUT	Check a test out of source control.
TEST.CHECKIN	Check a test into source control.

Using Mocks and Stubs

A particular concern for developers who are involved in unit testing code - especially legacy code - is to isolate the area of code that is being tested from any other pieces of code that it may call. This makes it possible to set up specific test conditions, or to remove dependencies on code that either has not been tested, not yet created or possibly has undesirable side effects when run under test conditions.

Imagine the following scenarios:

- You are testing a routine that calls a subroutine to request an input from the user. In this case, you want to force that input to a specific test value and not interrupt the test.
- You are testing a routine that calls a subroutine to send an email message. You want to test the message content but you do not want that message to be sent out.
- You are testing a routine that calls an error handling subroutine under certain conditions. You want to make sure that your coverage means that this routine has been called.
- You are testing a routine that calls a poorly-performing legacy subroutine and you do not want to hold up your test.
- You are testing a routine that calls another subroutine, and you want to check the arguments passed, but you do not want to change the called subroutine to write out the arguments.
- You are testing a routine that calls another subroutine, but you haven't written that second routine yet.

These situations are fairly common and so mvTest provides a neat means of handling them following on from the standard techniques found in other unit testing frameworks.

Mocks and stubs are temporary subroutines created for testing purposes. A stub generally only receives arguments to ensure that the test can continue, whereas a mock may also return specific arguments on demand so you can test how your routine will handle them.

mvTest Mocks

mvTest creates mock subroutines on the fly in response to the CreateMock function:

```
MyMock = CreateMock( subroutine_name, argument_list, original_file)
```

Where:

subroutine_name	is the catalog name of the subroutine to create.
argument_list	is a comma separated list of the subroutine arguments.
original_file	is the name of the original program file holding the subroutine.

The CreateMock function generates a mock subroutine with the specified arguments in the tsmock.bp file, compiles and locally catalogues the routine so that the calling routine will run the mock in preference to the real subroutine. The mock records the arguments passed and the number of times the mock has been called in the TEST_MOCKS file

For example:

```
! mock the email subroutine

EmailMock = CreateMock("SEND.EMAIL","Sender,Recipient,CC,Subject,Body","BP")

! Call my routine (won't send the email out now)
Call mySub
```

Checking the Mock

Once created the mock is accessible as mvTest object.

This exposes a number of simple methods that allow you to check the arguments that got passed and the number of calls:

```
Assert "Email routine should have been called", EmailMock.WasCalled()

Sender = EmailMock.Was("Sender")
AssertIs "Sender should be demo address", Sender, "demo@demo.com"
```

You can also stack the arguments that you want the routine to return. Here for example the script wants to forcibly return a specific argument from an input subroutine:

```
! set up the mock and its conditions
InputMock = CreateMock("INP","X,Y,LENGTH,ANS,HELP","")
InputMock.Want("ANS",2) ;* return 2 in ANS

! now call the routine being tested
Call mySub
```

Tearing Down the Mock

You can tear down the mock at any time explicitly by calling the Restore() method:

```
myMock.Restore()
```

If you do not tear down the mocks yourself, any mocks will be automatically torn down when the script runner completes its actions.

The method of tear down depends on the arguments you specify when creating the mock in the CreateMock function.

If you supply the original program file name, mvTest will automatically recatalog the routine using a regular catalog command:

```
CATALOG orig_bp_file itemname LOCAL FORCE
```

If you do not supply the original program name, mvTest will capture the catalog entry from the VOC file before the mock is created. If this is found, it will simply be restored to overwrite the temporary catalog entry. If not, mvTest will assume that the routine is globally catalogued and will delete the local (direct in UniData) catalog pointer.

MOCK Object Methods

The following methods can be used on a Mock object. Where a subroutine argument is required it is specified by name to ensure the tests are legible:

Clear()	This clears down the counters and arguments captured for a mock without tearing down the mock itself.
Called()	This returns the number of times the mock routine has been called.
Check(arg,specification)	This performs an mvTest Check() function on the value passed into the requested argument.
Restore()	This tears down the mock and restores the original routine.
Want(arg)	This sets the desired return value of a given argument.
Was(arg)	This returns the value that was sent for a given argument.
WasCalled()	Returns true if the routine was called at least once.
WasNotCalled()	Returns true if the routine was never called.

Setting Remote Locks

When performing testing of multi-user software locking is an essential ingredient, as is the ability for software to operate correctly when another user has locked records of interest.

mvTest supports lock testing through the static LOCK object.

The LOCK object invokes the TEST.LOCKDAEMON phantom and messages it to set, release or check locks on its behalf, so that the locks appear to be set by another user.

The LOCK object provides a convenient abstraction of this process.

Configuring the LOCK Phantom

The TEST.LOCKDAEMON phantom acts as a socket listener, waiting and acting on requests from the test script runners.

The LOCK Phantom is configured through the TEST_CONFIG MVTEST.INI record as follows:

```
[Lock]
Port=port number for the phantom to listen against.
Address=address for the phantom to listen on.
Timeout=maximum socket read time in milliseconds
```

These will default to port 40003 on the localhost (127.0.0.1) with a 10 second timeout if no configuration data is supplied.

LOCK Object Methods

The LOCK Object supports the following methods:

Set(FilePath, RecordId)	Sets a record lock
Release(FilePath, RecordId)	Releases a record lock
Check(FilePath, RecordId)	returns TRUE if the phantom has locked the record
Clear()	Clears all locks set by the current user.
List()	Lists all locks held by the phantom
Close()	Closes the lock daemon

Note that the file path is required for correct locking operation, as the phantom may be running in another account than that which is being used for testing. On UniVerse and UniData you can use the FILEINFO() function to return the file path for an open file.

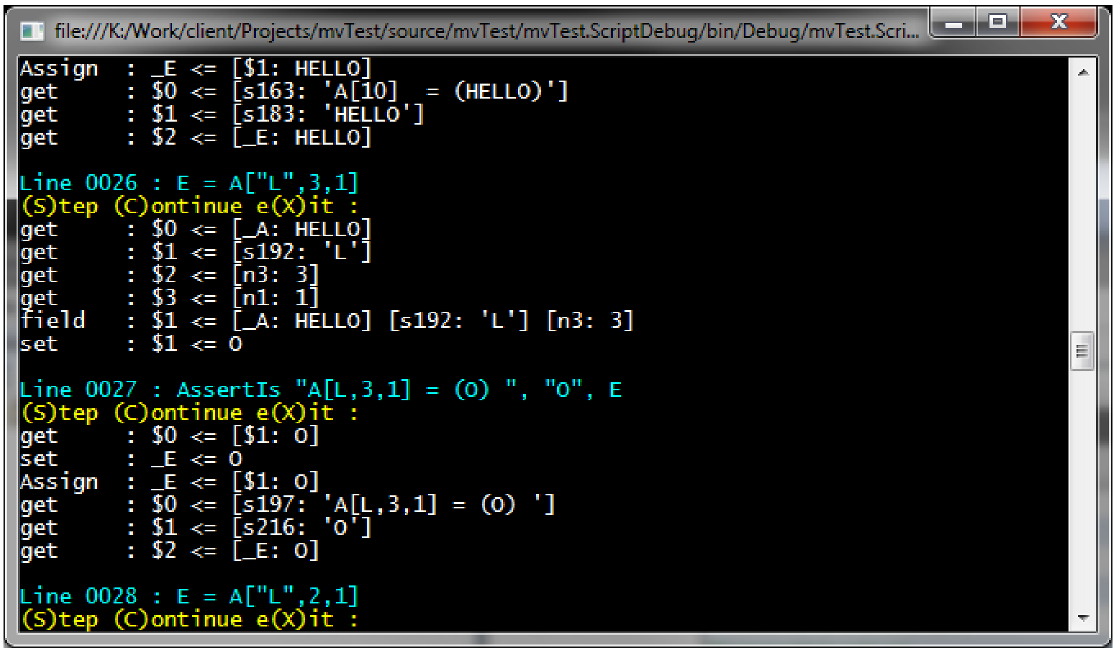
Debugging your Scripts

From version 1.3 of mvTest you can debug your scripts as they run through the client side script debugger. This displays the source and script object being executed and allows you to step through these.

The Script Debugger runs in a separate window as a client application to prevent it from directly interfering with the mvTest client. It acts as a socket listener running on port 40011 on the client.

If you wish to enable debugging you must start the debug client explicitly before running your tests. That way, if you accidentally leave a DEBUG statement in them (haven't we all at some point) it won't break any integration or unattended testing.

Note that if you have scripts compiled in a previous release (before 1.3) you will need to recompile them before they can be debugged in this way.


file:///K:/Work/client/Projects/mvTest/source/mvTest/mvTest.ScriptDebug/bin/Debug/mvTest.Scri...

```
Assign  : _E <= [$1: HELLO]
get     : $0 <= [s163: 'A[10]  = (HELLO)']
get     : $1 <= [s183: 'HELLO']
get     : $2 <= [_E: HELLO]

Line 0026 : E = A["L",3,1]
(S)tep (C)ontinue e(X)it :
get     : $0 <= [_A: HELLO]
get     : $1 <= [s192: 'L']
get     : $2 <= [n3: 3]
get     : $3 <= [n1: 1]
field   : $1 <= [_A: HELLO] [s192: 'L'] [n3: 3]
set     : $1 <= O

Line 0027 : AssertIs "A[L,3,1] = (O) ", "O", E
(S)tep (C)ontinue e(X)it :
get     : $0 <= [$1: O]
set     : _E <= O
Assign  : _E <= [$1: O]
get     : $0 <= [s197: 'A[L,3,1] = (O) ']
get     : $1 <= [s216: 'O']
get     : $2 <= [_E: O]

Line 0028 : E = A["L",2,1]
(S)tep (C)ontinue e(X)it :
```

From the debugger you can:

Press S or <Return> to step through the code.
Press C to continue to the end of the script.
Press X to exit the script.
Press V to enquire on the current value of a variable.