



ABN: 99 070 946 986
www.qualtech-int.com.au
email: info@qualtech-int.com.au

Level 2, 222 Pitt Street
Sydney NSW 2000
Phone: (02) 9264 0055
Fax: (02) 9264 0033

Level 28, 303 Collins Street
Melbourne VIC 3000
Phone: (03) 9614 8555
Fax: (03) 9614 8666

QualTech International

WinRunner User Guide

A white paper written by Matt Crump and Ivan Maclaine

Contents

[Installation](#)

[Setup](#)

[GUI Maps and the Naming Convention](#)

[Custom Functions](#)

[Compiled Modules](#)

[External Functions](#)

[Creating Test Scripts](#)

[Running Test Scripts](#)

[Interpreting Results](#)

[Tips and Tricks](#)

Installation

The way in which WinRunner is setup can have an enormous effect on the performance of your system. It is important to decide up front if your internal network (Intranet) can handle the bandwidth required by a networked WinRunner installation. Failure to do so will result in a very slow and tedious setup, leading to an overall loss of productivity and general annoyance. Unless you plan to be changing test machines frequently (see tips and tricks) we suggest installing WinRunner on every machine with the stand-alone option. This places a local copy of all the binary files on the machine's hard drive, rather than on a network file share or server. Installing to the network saves hard drive space, but this is more than made up for by the increased speed.

Installing as stand-alone does not mean you cannot share a test repository between instances of WinRunner. A networked fileshare can be used to store WinRunner test cases and results. We believe this is the most effective way of implementing your testing suite.

Setup

Once you have WinRunner installed, you need to make a few decisions as to how and where you will store your test cases and results. The best place to store your scripts and associated data is on a shared or network drive. This drive must have read/write access by everyone who is going to be using WinRunner. Be warned that you should place your test scripts as close to the root directory of the drive as possible. Problems can arise when using non-Windows based back ends. For example if you use a VAX system as your file server, any more than 8 levels deep in the directory structure and your data will be untouchable.

Both WinRunner and TestDirector place restrictions on the length of a path. If either a direct pathname or an UNC path is decoded to greater than 60 characters, then access will fail.

Once you have WinRunner setup and running, you must start to think about how you will implement any custom changes to the way in which WinRunner starts up. The most obvious place to implement any changes is in the `tsl_init` file. This would appear as a good idea, but after installing several product updates and service packs, it became apparent that this file is overwritten with the new version, thus destroying any of your changes. A much better idea is to create your own init file and name it `myinit` (or something similar). Then make a single line change to the original `tsl_init` file by adding:

```
call "myinit" ();
```

By doing this, any future version will only destroy the call to your script, which can be easily added again at a later date.

GUI Maps and the Naming Convention

There are two methods to generating test scripts. The first is to map out the entire application under test with the GUI editor before scripting begins. The second method is to learn each screen, as it is needed. Both methods have their merits, and either method will do. The important point to learn is how you name your objects and windows. Thus a naming convention must be put in place. It is important to decide on this from the start, and not change it throughout the entire testing cycle. Failure to do so will lead to test

discrepancies, causing confusion.

Ideally when you learn a window, you should rename all the objects to something intelligent. Although WinRunner attempts to automatically name the objects, if the object has no text associated with it, WinRunner has to give it a generic name. This normally bears no resemblance to the actual meaning of the object. The preferred method of naming learned objects is by preceding them with the name of the object. For example a push button labelled 'Interest' becomes ButtonInterest. This can be summarised by the following table:

<i>Object</i>	<i>Naming Convention</i>
Push button	ButtonName
Edit field	EditName
Listbox	ListboxName
Radio button	RadioName
Combobox	ComboName
Checkbox	CheckName
Window	NameWindow

By adhering to this naming convention makes it very easy to create a test script without continually having to refer to the GUI map, which can be a slow process if the GUI map is large.

Custom Functions

The easiest way to make your test scripts maintainable is to use high level custom functions. You should start with some kind of *boot* or *login* function. When called this function will open the application under test and navigate to the main startup screen. This will be the base screen for all testing to commence.

Once the base screen is accessible, it is advisable to create functions, which navigate to other main screens of interest. These functions will first check to see if the main screen is visible, and if not, will load it and proceed to the desired screen. The depth of screens that you decide to create custom navigation functions for will depend on the size and time frame of your project.

Once navigation has been taken care of, you need to start looking at the kind of things your scripts are going to be testing. Look for trends and patterns amongst what is being tested. Such occurrences make good candidates for being written as functions.

Compiled Modules

When you have a library of functions, it makes sense to break them into manageable blocks. These blocks are called compiled modules. A compiled module is made up of one or more functions. To access a function in a compiled module, the compiled module must first be loaded using LOAD("").

It also makes sense to organise your custom functions into appropriate compiled modules. Examples are a navigation module, and a text based module.

External Functions

As with compiled modules, external functions in the form of Dynamic Link Libraries (DLL) can be accessed by using the LOAD_DLL command and defining the function as an EXTERN.

Creating Test Scripts

The best way to create a script is to get the skeleton right first. This means coding a brief script that includes:

Any house keeping code.

Application under test startup/login.

Navigation to the screen to be tested.

Setup any relevant data.

Perform the necessary actions.

Check the results.

Report the results.

The house keeping code includes initialising any global variables, loading compiled modules and defining extern functions. This can't be done in a custom function, as the function cannot be seen by the compiler if it is not first loaded, hence a paradox is created.

The startup/login code should ideally be in a navigation compiled module.

Running Test Scripts

Scripts can be run in either of two ways. The Mercury recommended way is to link your tests to Test Director and control them from there. This is great in a perfect world, but due to the way in which Test Director works, a lot of the information gathered from WinRunner is not accessible. This is fine when your scripts are running perfectly, but unfortunately this is not always the case.

By running your scripts directly from WinRunner, you are able to view exactly where any problems have occurred, either with your script or changes in the application under test.

Interpreting Results

The results of running one or more tests can be quickly determined through the WinRunner

Reports window. This window breaks down the results of the test run into three sections.

The first section on the left displays all the tests that were run during the specified test execution (defaults to the last test run). The tests are presented in a hierarchical structure, identical to the way in which they were called. So if test b is called from inside test a, then test b will be displayed as being a *child* of test a. The primary use of this section of the display is to select individual tests so the results of each can be studied in detail.

The second section appears at the top of the screen, and summarises the result of the test that has been selected on the left. This section gives an overall pass or fail for the test, as well as detailing the total number of bitmap and GUI checkpoints. It is also possible to view these checkpoints individually, showing which passed and which failed. It is worth noting that a single failure in a test causes the whole test to be given the status of failure.

The third section of the screen contains the bulk of the information on the test run. It gives a line-by-line description of what occurred in the test, and the outcomes of any checks performed. Not all lines will be represented in this display, only lines that contribute to the result of the test, or provide useful information about the running of the test.

The lines presented here are colour coded. Green lines represent checkpoints that passed, red lines checkpoints that failed, and black lines provide added information. It is possible to double click on most lines in this display to bring up a window that provides more information about the given event. For example, double clicking on an end GUI checkpoint line brings up a window displaying the expected results and the actual results for the object(s) in question. This allows a user to quickly determine why the failure has occurred, and whether the failure is in the application under test, or a deficiency in the WinRunner script. If it is determined that the script is at fault, due to, for example, an improvement in the application under test that wasn't accounted for in the script, the update button can be pressed to copy the actual current result into the expected result for the checkpoint, so that it will pass the test from now on.

Tips and Tricks

Don't change machines often. If you change machines often, then it is highly recommended that you use the network install. This has its own drawbacks, mentioned above. Each time you reinstall WinRunner on a new machine, you will have to repeat the steps listed in the *Installation* section.

Precede all custom functions with a lower case 'f'. For example, fLogin();

This will enable you and others to quickly ascertain whether or not the function is custom.

It is preferable to record actions and to code with the function generator than to write code by hand. The first two methods minimise syntactical errors, and can even help reduce logical errors. Although hand coding can often be quicker initially, this is usually outweighed by the extra time spent debugging scripts.

Where possible, keep the length of scripts to a minimum, so long as they don't lose meaning as being useful test cases. Shorter scripts are more robust, and when problems do occur, more easily maintained than longer scripts.

Have one or two people responsible for creating an initial GUI map, and ensure that it is comprehensive, before general script creation commences. This prevents the situation

where two users open the GUI map, make changes, and then resave the GUI map. The second user will overwrite the changes of the first user. This situation becomes unworkable when a number of users need to constantly modify the GUI map.

Don't run with scissors!