## Tip-of-the-Month Website Submissions

The Allegiance Group website, www.allegiance.com, 2000-2001

## Handling Application Message Dialogs in SilkTest

Using an automated test tool such as SilkTest, instead of trying to declare all message dialogs used in your application, try using dynamic instantiation instead. That is, don't declare message dialogs in your script, instead write a generic function that issues a call to Desktop.GetChildren() which will return all the windows on the desktop. Then code a loop using the window.GetClass() method and check for windows of the Dialogbox class.

Once you have identified the name of the message dialog you can then look for static text objects, which contain the message text you are looking for. You may incur a small runtime performance penalty using this technique; however, the benefits are reduced script maintenance and improved test reliability.

Using this approach, your test scripts will be able to capture unexpected or unknown messages that could otherwise escape detection or cause your scripts to fail.

## Handling Application Message Dialogs, Part II

Occasionally, an automated test script must detect a series of message dialogs that can be displayed by an application under test. These message dialog types can be error messages, informational messages or prompts for the user to provide additional input data. During normal system behavior, the exact display sequence of these dialogs is known beforehand. Under these normal conditions, the automated test script will detect the dialogs successfully using a top down, serial coding approach. However, script problems can occur if these dialogs appear out of order, or if unexpected dialogs appear. The problem is only compounded if any of these dialogs are modal. One technique that you can use to handle this situation is by looking for dialogs concurrently. This is done in 4Test using the parallel statement or alternatively with the spawn/rendezvous statements. This technique can be implemented using the following code snippet as a template:

```
parallel {
        while (TRUE) {
                if (dialog1.Exists ())
                        critical {
                                // specific code to handle dialog 1.
                        }
                        break
                else
                        if (TimerValue (hWaitTime) > 30.0)
                                break
        }

        while (TRUE) {
                if (dialog2.Exists ()
                        critical {
                                // specific code to handle dialog 2.
                        }
                        break
                else
                        if (TimerValue (hWaitTime) > 30.0)
                                break
        }
}
```

The above code requires a timer to be created and started prior to the parallel statement. The critical statement prevents the indented block of code from being interrupted by another process. Each dialog must be processed without interruption in order to prevent conflicts with other dialogs that may appear. By using this technique, you can increase your script's probability of catching rogue message dialogs.

## Using Parameters to Control Test Execution

One of the primary objectives of a well-written test library is that a tester can execute any given test script without having to modify the code tree.

In SilkTest, this means that Test Scripts (.t) and Include Files (.inc) remain intact, whereas Testplans (.pln), Suites and Option Files are modified according to individual test requirements. SilkTest provides a number of features that a test developer can use to achieve this level of script generalization. These features involve the use of compile and run-time arguments, text files, INI files, and environment variables.

Run-time Arguments can be passed to test scripts by entering text in the Arguments field in the Runtime Options window. Alternatively, these arguments can also be passed to test scripts from an Organizer testplan or suite. Compiler constants can be set in the Compiler Constant window invoked by a pushbutton from the Runtime Options window. Compiler constants act as global variables and can be referenced from any test script or include file. Compiler constants are constants and cannot be modified by a running test script.

Run-time test parameters can also be set in INI files. SilkTest provides built-in functions that provide straightforward access these files.

Text files are another way of maintaining test data that can also be accessed using built-in functions.

Finally, SilkTest allows environment variables to be initialized and retrieved during test execution. These environment variables are accessed using the SYS_GetEnv and SYS_SetEnv built-in functions. Environment variables persist until the current running instance of SilkTest is terminated or restarted.

## Creating Hierarchical Levels in SilkTest Result Files

Most SilkTest coders are reluctant to enable the "Print Agent Calls" feature in the Runtime Options dialog box for production test runs because the results file becomes too verbose and difficult to read. On the other hand, this debugging feature is very useful to have for troubleshooting bizarre script behavior.

This tip of the month will describe a simple way to hide these Agent calls by creating hierarchical levels within SilkTest results files.

SilkTest provides two functions, ResOpenList and ResCloseList, which allow hierarchies to be created in results files. The trick is to code these statements within each function, method, and appstate called by your test scripts. The recommended way to do this is to code a ResOpenList prior to the first executable line in a function, method or appstate followed by Print and ResCloseList statements at the end or immediately prior to a return() statement.

For example, the following statements would be coded for a ParseCSVLine function:

```
//<Variable declarations>

ResOpenList (">> Function::ParseCSVLine ({sInputLine}, {iStart}, {iEnd})")

//<Executable statements>

Print ("<< Function::ParseCSVLine")
ResCloseList ()

return (lsData)
```

By implementing this technique your results files will exhibit the terseness you desire, but the detail can be readily exposed by using the 4Test editor's expand and collapse feature.

## Using Recursion in 4Test

Recursion is a technique that allows you to solve large problems by reducing them into smaller subproblems, which have a similar structure to the original problem and are somewhat simpler to solve.

In programming, a recursive procedure is a procedure that calls itself from within the procedure body, thereby creating an additional activation of the procedure during the lifetime of the first activation. Recursion is a very powerful but undocumented feature in SilkTest's 4Test scripting language.

Recursive functions can return values in 4Test by one of two different techniques—one is to use the return statement and the other is to use an `out` or `inout` argument type in the function declaration. The `out` and `inout` argument types force 4Test to pass the argument by reference instead of by value which is the default.

For functions implemented using the `return` statement, care must be taken to terminate the recursion otherwise the return value may be overwritten. On the other hand, the out/inout argument technique is more forgiving if unnecessary recursive calls are invoked.