# Part 5 : Testing and Debugging

## Methods of Testing

The most obvious means of testing is, of course, simply tapping in examples and checking that the expected answers appear. The trouble with this method is that it is tedious. It is likely that testing will need to be done many times during debugging and redoing a large number of tests will stretch the patience of most people. Still, it is a useful quick check for a particular operation.

A second more attractive method is to have some built-in self testing that can repeatedly run the same tests, as if you were keying in the sequences yourself. A data source of sequences could be read, with an associated result and the sequences fed into the input function of the javascript, and the resulting display compared with the specified expected result. In this way tests can be built up as functionality is added, and the previous tests rerun to ensure that old functionality hasn't been broken. Therefore it is beneficial that the test facility is made available early on in the cycle so that it is an aid to development and debug, as well as a method of ensuring accuracy. Below a method for implementing such a scheme is described. However, we still don't know what tests to run, even if we have a method of running them. We'll look at this now.

## Sources of Test Data

The best source of test data is the manual for the calculator you are simulating. This will have examples of all the features of the calculator and the expected results. The data will be presented in different styles for different manuals, and some features may be described in words only (e.g. what happens on an error condition), but all these should be translatable into a set of key sequences---you may have to make up some numbers if none are specifed. By testing all the examples in the manual, the resulting simulator will have captured the best part of the particular features for the calculator, and a quality model will have been made (once all the tests pass). However, what is sometimes missed in a manual is what happens when a non-standard sequence is keyed in, as will happen eventually if the model is used in earnest by someone. This brings us to the second source of test data.

Having the actual calculator you're modelling to hand is an invaluable tool for generating new tests. Being able to key in odd sequences and seeing what happens allows you to capture much more of the behaviour of the calculator than through the manual examples alone. You still have to think of new sequences not covered by the manual, but by using the calculator regularly, and making unintentional sequences, you can add the results to your tests. You may feel that undocumented features of a calculator are meant to have undefined results, and that if these differ between the calculator and the simulator this doesn't matter. You may be right---it depends on what level of accuracy you want, and what you're trying to convey to a potential user of the model. You'll have to decide that for yourself.

A third method, which is less effective, but perhaps the only available method, is simply to invent tests to try and cover all the features of the calculator with at least one example. This will test that what you intended in the design is what you have implemented, but won't test for accuracy of modelling. The other trap one can fall into with this method is using the model to determine what result is expected---especially if you are constructing test data for the built-in self test code. Try not to do this, but find another calculator to calculate the expected values. This methods maybe the only option you have, but it is unsatisfactory, and it may be worth asking on the internet if someone has the relevant manual, and would scan it in for you. (You could try a post on the Vintage Calculators Forum.)

## Automatic Test Code

In this section I will briefly describe the testcode employed in the calculator simulators I've written, and was used in the example calculator. The basic strategy is to have a string containing a number of test sequences, followed by an expected result. A function reads each sequence, calls the key_pressed() function and compares the expected result with the actual calculator result, and logs an error if they are different. The test sequence and result combination needs to be delimited in some way so that the function knows when a new sequence starts, and where the result is. In this code the basic test is "<input sequence> : <expected result> @". The colon marks the end of the sequence and start of the expected result, whilst the 'at' marks the end of the test, beginning of a new test. Although the tests will be stored all in one big string, we will put each test on a separate line and concatonate the strings (with +) so that the test number and line number are equal making debugging easier. The testdata string and test function are shown below.

```
testdata = "o:0.@" +                                    // initialise
"15.3 + 1.89 =          : 17.19 @" +                    // Add
"15.3 - 1.89 =          : 13.41 @" +                    // Subtract
"15.3 * 1.89 =          : 28.917 @" +                   // Multiply
```

```
    "15.3 / 1.89 =          : 8.0952381 @" +                // Divide
    "15.3 + 1.89 i =        : 13.41 @" +                    // change sign
    "15.3 s + 1.89 =        : 5.8015214 @" +                // square root
    "15.3 * 1.89 %          : 0.28917 @" +                  // percent
    "22.5 c 15.3 + 1.89 =   : 17.19 @" +                    // clear entry
    "1.89 m 15.3 + r =      : 17.19 @" +                    // M+ and recall
    "3.78 M 15.3 - r =      : 17.19 @" +                    // M- and recall
    "55 + 99 =              : 154. @" +                     // Trailing decimal
    "o:0.";                                                 // last line

    // ------------------------------------------------------
    function test() {

        var answer = false;
        var testnum = 0;
        var failures = new Array();
        var fidx = 0;
        var testchar = "";

        for (tidx = 0; tidx < testdata.length; tidx = tidx + 1) {

            testchar = testdata.charAt(tidx);
            if (testchar != ' ') {
                if (testchar == ':') {
                    answer = true;
                    expected = "";
                } else if (testchar == '@') {
                    answer = false;
                    result = disp;
                    if (parseFloat(expected) != parseFloat(result))
                        failures[fidx++] = testnum;
                    testnum++;
                }
                else if (answer == true)
                    expected = expected + testchar;
                else {
                    if ("0123456789.+-*/=%simMroc".indexOf(testchar) == -1) {
                        alert("Invalid input character ("+testchar+") in test sequence\n"+
                              "at line "+testnum);
                        return;
                    }
                    key_pressed(testchar);
                }
            }
        }

        testchar = "";
        for (tidx = 0; tidx < fidx; tidx++)
            testchar = testchar + " " + failures[tidx];

        if (fidx > 0)
            alert (fidx+" failures at lines:"+testchar);
        else
            alert ("All "+ (testnum-1) + " tests pass");

    }
```

As the code above only uses JavaScript features that have been described in earlier sections, I won't decribe it in great detail. The main features to look for are that normally the characters are read in and fed straight into key_pressed(), except that spaces are removed. If ':' is encountered, the variable 'answer' is set true, and the following input is read into a variable 'expected'. If '@' is encountered, 'answer' is cleared again, and the result (from disp in the main code) is compared with 'expected'. If a failure is encountered, the failed test number is added to the end of the array 'failures', so

that all failed tests are logged. At the end of the tests, failure (or pass) information is displayed via an 'alert', which simply pops up a message box with the displayed string---in this case the list of failed tests, or a pass message. Notice also that the testdata string's 'zeroth' test is 'o:0.', used to ensure the calculator is in a known state. A user may have pressed some keys before the test is run. The only other thing to do is have a means for running the test function. Perhaps the easiest way is to add a trap in the keyboard() function, where, say, pressing 't' runs test(), else the character is passed to key_pressed(). You can popup the calculator here, click on the image and press 't', and you can run the tests.

## Debugging

The test code above will help in finding errors in implementation. If there are failures, a list of the tests that fail will be displayed so that you will know what functions aren't right. You can comment out tests if you want to concentrate on only one failure. I tend to work from the top down, sorting one issue before moving to the next, whilst still ensuring previously working tests still pass. The above code won't find basic syntax errors for you though.

JavaScript is an interpreted language, and so the first time you find out you typed some illegal code in, is when it is executed. Usually your browser will silently skip illegal code and you have no idea that something is wrong. For Internet Explorer users you can get more information about failures by reconfiguring your preferences. Other browsers probably have similar features, I just don't know how to access them (drop me a line, I'll add the information here!).

What follows refers to Internet Explorer version 5.50, though I've no reason to suppose it is much different on other recent versions. From the menu select 'Tool->Internet Options...'. Select the 'Advanced' tab on the resulting popup. In the 'Browsing' subcategory, make sure that 'Disable script debugging' is unchecked and 'Display a notification about every script error' is checked. Click 'OK'. Now, if a bad piece of javascript is encountered, a popup window is displayed with some information about that error. The most useful piece of information is the line number. Often this indicates the exact place of error, though beware that it may be before this (as happens when, say, a string has no close quotes and this is picked up at the next set of quotes). It's not perfect but its better than fumbling in the dark. You might want to turn the debugging off when not working on your simulator. Many pages on the internet have bugs (I know---unbelievable!) and one can get fed up with each error causing a popup to be displayed.

One final option open to you is the use of an 'alert'. If you suspect a problem in a certain area, and want to know what the state is before or after something has been run, you can construct an alert to display relevant state. For example, suppose at the end of reduce() we want to know the value of all the main registers, x, y and m. We can do this by adding the following line as the last line of reduce().

```
alert("x = " + x + " y = " + y + " m = " + m);
```

Whenever reduce() is called, a popup appears with a message displaying the register values. You can place these at various locations to track down where a defect occurs.

"How to write a calculator simulator"
<- Prev Page Next Page->