Part 4 : The JavaScript

JavaScript Introduction

It is not possible, of course, to give a full tutorial on JavaScript in this article. Various links in the appendix will refer you to more documentation and example code. The general structure is like 'C' or 'C++', and if you know C the code will look familiar. Of the JavaScript specific constructs, those relevant to this model will be briefly discussed now. In this discussion the 1.1 version of JavaScript is referred to. Later versions improve on some of the limitations, but as stated before we will stick to 1.1 for maximum compatibility.

The first thing a C programmer would notice about the javascript code is that all the functions and procedures have the key word 'function' before them, and no return type. If any value is returned at all (using 'return'), its type is implied by the type of the returned variable. This also brings us to a second key difference, there is almost no typing of variables (and you thought C was bad!). Variables do not have to be declared (though we will declare them in this example), and the type they take on is dependant on the data type they are assigned. A variable is declared with 'var <varname>'.

We will be manipulating strings extensively, and JavaScript has some built-in features for this. A string literal can be enclosed in single or double quotes. In this model an arbitrary convention is used where characters (strings of length 1) are enclosed in single quotes and strings are in double quotes. The length of a string can be determined directly. If a variable 'str' contains a string, then its length is 'str.length'. This also works for string literals (e.g. "1234".length), as does that which follows. To determine if a particular character is in a string, and its location, you can use something like str.indexOf('+'). This will be -1 if it is not found. To get the character at a particular location then str.charAt(0) can be used. To get a substring a statement such as str.substring(1, 3) is used. We can concatonate strings together using '+'. E.g. "hello " + "mum" equals "hello mum". Numbers can be converted to strings as "str = x.toString()".

Numbers are floating point. Other than that they are treated as you'd expect. To manipulate numbers beyond the basics, a set of built-in 'methods' are provided by a 'Math' object. So, to find the square root of a variable x, use Math.sqrt(x). The only other Math method we will use in this model is Math.round(). Refer to the JavaScript links for all the methods available. Changing a string to a number is quite useful, and this is done with the "parseFloat(str)" built-in function.

The last thing to introduce is the instantiation of the code in the HTML. We have two choices---embed or reference. We will use the former, though the latter is more usual as it allows more than one page to reference the code. To embed code the <script> HTML directive is used eg.

```
<script type=text/javascript>
</script>
```

The javascript is placed bewtween the script tags. To reference a file containing the script use (for example, in a file 'calc.js'):

```
<script type=text/javascript src="calc.js">
</script>
```

So we just about have enough to proceed.

General Structure

We already know from Part 3 the names of two functions we must have, namely keyboard() and key_pressed(). These are the input interfaces for the HTML. The output (i.e. display) also needs an interface, and we will have a function called update_display() that is responsible for changing the display graphics---how it does this is described below. We could implement everything in just these three functions, but we will have some more refinements than this in the final model.

As well as the functions we need some state. The calculator has registers that we will need to reflect. So some global variables are declared

```
var x;
var y;
var m;
```

The x register is an accumulator where the results are placed. The y register is the input number and the m register is the memory. As well as the models of the calculator's register, we need internal state to control the operation.

```
var disp="";
var is_new_num = true;
var is_decimal = false;
var last_op = "";
var error = false;
```

The disp variable is a string that holds the current value to be displayed as a string rather than a number. It is the variable passed to update_display(). The flag is_new_num is set whenever a numeric input needs to start a new number input, as opposed to appending to a number already being input. Similarly is_decimal is set if the decimal point key has been pressed. The last_op variable holds the last operational key (i.e. one of '+', '-' '*' or '/') or is null if none active. Finally, the error flag is set on overflows and other errors. It is used by update_display() to display an error state, and also to prevent all other inputs apart from 'ON/C'. These variables also have initial values set at declaration.

We also declare variables to hold the images. This will make referencing the images simpler, but also forces the HTML to load the images immediately. In general a variable is declared and assigned some space, and then the source file for that image variable set to point to the relevant file.

```
lcd0 = new Image(20, 45);
lcd0.src="images/lcd0.jpg";
```

The 'new Image(20,45)' creates an image structure of the declared size, and lcd0 is assigned to it. The src variable of the new structure is then assigned to the image filename. This is done for all the images we will use.

Changing the Display

The above description of creating an image variable gives a clue as to how we can update the display. In Part 3, it was explained how the display was constructed in an HTML table. The <img> constructs for the display were given a name attribute---"d0" to "d7". If we change the src attributes' value of the img, then we can alter the display. So, for example, the src attribute of the image named "d0" is document.d0.src. Similarly for the others. So the update_display() function calculates the new image file names needed for each display segment based on its input string, and then updates the relevant images in the HTML.

To do this an array is created to contain a list of image src filenames, and then after assigning the names for each segment the 'document' references updated to display the new value. A 'while' loop is employed to march through the segments. A fragment of the code is shown below.

```
function update_display(dspin) {
    var disp_array = new Array();
    var dot_active = false;
    var dsp=dspin;
    var lcds=0;
    var idx=dsp.length;

    while (lcds < 8) {
        idx--;
        digit = dsp.charAt(idx);
        if (digit == '.')
            dot_active = true;
        else if (digit == '-') {
            disp_array[lcds] = lcdminus.src;
            lcds++;
        } else if (digit && "0123456789".indexOf(digit) != -1) {
            if (dot_active)
                disp_array[lcds] = "images/lcd"+digit+"dot.jpg";
            else
                disp_array[lcds] = "images/lcd"+digit+".jpg";
            dot_active = false;
            lcds++;
        } else {
            disp_array[lcds] = lcdoff.src;
            lcds++;
        }
    }
    document.d7.src = disp_array[7];
                  .
                  .
    document.d0.src = disp_array[0];
}
```

The display is right justified, so our index (idx) is assigned to the rightmost digit of the input string (dspin), and will count down. An array is created (disp_array) using 'new Array()', and the input assigned to a local variable (dsp) so that we can alter the value. The 'lcds' variable is used to index the array, and is initialised to 0. So the while loop continues until all 8 segments are processed. A character is pulled from the string and assigned to a local variable digit, which is then tested for various values. If it is a decimal point, then dot_active is set true. If it is a minus, then the array indexed by lcds is set to the src of the minus image variable. For numeric characters the array element is set dependant on two things. Is dot_active true, in which case use the dot versions of the images, and what the value of the digit is. Strings are constructed from this which will refer to the correct file and assigned to the array. If digit is none of the recognised values, the array is assigned with the lcdoff source. Note that the idx value is decremented for each loop except if the digit is '.' which is simply consumed as it only modifies a digit display to a dot version, but does not move a segment on.

Not shown in the fragment is the case where the 'error' flag is set, or ensuring that the input string has a decimal place (even if at the end for integers). These are small enhancements, and you can see the full code via the link at the bottom of the page.

Processing input

The input is processed by the two functions keyboard() and key_pressed() as explained above. The keyboard() function's job is simply to extract information from the event structure that is passed in so that key_pressed() can be called with an input in the right form.

```
function keyboard(e) {
    ky = String.fromCharCode(e.keyCode);
    key_pressed(ky);
}
```

Without going into too much detail, the event structure in 'e' has a keyCode value (the unicode value of the key pressed) which we can convert to a single character string with the built in method of 'String.fromCharCode()'. The main input processing function key_pressed() can then be called with the extracted character, so that all input is processed in the same way.

The key_pressed() function is a big 'if' statement (we might have used 'case', but that's not available in JavaScript 1.1). The input character falls into one of four classes. A number (including decimal place), an operation (+, -, *, or /), a completion (= or %) and the rest. After processing, a call to update_display() is made to display a new value (though it could be unchanged). So the general structure of key_pressed() is as shown below.

```
function key_pressed(key) {

    if ((key && ".0123456789".indexOf(key)) != -1) {
        // Process number
    } else if ("*/+-".indexOf(key) != -1) {
        // Process operation
    } else if ("=%".indexOf(key) != -1) {
        // Process completion
    } else {
        // Process other
    }

    update_display(disp);
}
```

In the first branch of the tree (numbers), we generally want to keep adding the input to disp (disp = disp + key). If the number is not a decimal (is_decimal is false), we keep the decimal at the end. If the display is full we ignore the input. When a decimal is input is_decimal is set true. If a decimal is the first number (is_new_num is true) then we imply a leading 0 (disp = "0."). After processing the input number, the numeric value is assigned to y (y = parseFloat(disp)) so that y always contains the latest input value should an operation key be pressed. The variable is_new_num is always cleared on a numeric value, as we must be in the middle of an input.

For operation keys, in a sequence like "2 + 3 = ", the operation key (+) simply copies the input (in the y register) to the x register, the actual operation being delayed until the completion key (=). So the pending operation must be stored until completion. This is what last_op is used for. After any operation a new number key must be a new input, so is_new_num is set true, with is_decimal false. However, for a sequence such as "2 + 3 - 1 =", the second operation (-) must effect a completion as if the equals had been pressed. So in this case, where last_op is not an empty string, we don't copy y to x, but do the last operation on x and y. A function reduce() is called with the last_op value to do this (see below), and then disp updated with the resulting value in x.

Completion is very similar, except reduce() is always called to update x (unless mistakenly pressed before an operation input). If '%' is input instead of '=' then y is divided by 100 before calling reduce.

The remaining input processing has three catagories: modifying y, clearing and memory register manipulation. The 's' key square roots the y register, and disp is updated. Before disp is updated, y is rounded to a value that fits on the display by calling a function rnd_to_display(). This is explained below. Also, square rooting negative numbers needs to be detected, and the error flag set. The 'i' key inverts y (i.e. multiplies by -1) and disp updated. The 'c' key clears y to 0, sets disp accordingly and sets state for a new number. The 'o' key initialises all the state back to the original values. For the memory operations, 'm' simply copies the value in the m register to y, and disp is updated. The key 'm' (add to memory) and 'M' (subtract from memory) simply update the variable 'm' by adding or subtracting y.

The reduce() function gathers together the actual math operations of the calculator into one place (it's called from two separate places). It's basic function is simple, in that the x register is updated with the y register via the selected operation passed in as op. A fragment is shown below.

```
function reduce (op) {
      if (op == '+')
          x = x + y;
      if (op == '-')
          x = x - y;
      if (op == '*')
          x = x * y;
      if (op == '/') {
          x = x / y;
      }
  }
```

In addtion to this, we need to trap division by zero and flag the error. Then the resulting value of x must be rounded so that the result only has as many decimal places as will fit on the display. Finally x is checked that it lies within the maximum and minimum range of the calculator, and error set true if it is outside.

The rounding of x is done via another function rnd_to_disp(). Like reduce(), this is gathered into a function as it is called from key_pressed() as well. Here we multiply x by a factor so that all relevant decimal points would become whole numbers, call Math.round() to round to nearest value, and divide the same number of places down again.

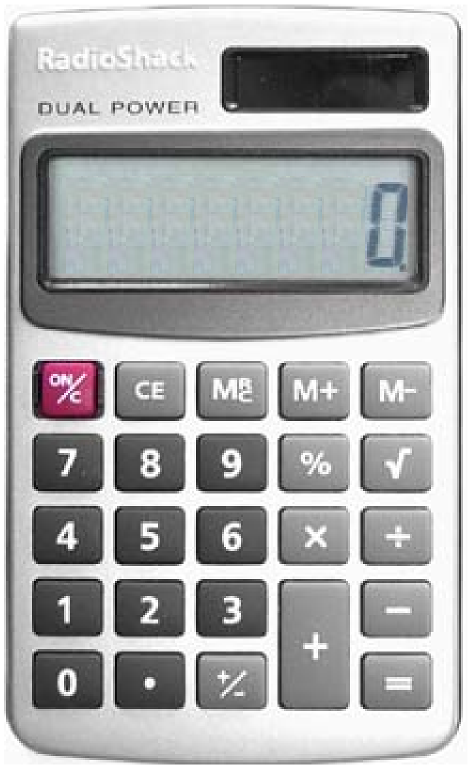
```
function rnd_to_display(val) {
      var rnd_factor;
      var result = (val < 0) ? -1 * val : val;

      rnd_factor = 10000000;
      while (result >= 10) {
          result = result / 10;
        rnd_factor = rnd_factor / 10;
      }
      if (val < 0)
          rnd_factor = rnd_factor / 10;
      result = (Math.round(val * rnd_factor))/rnd_factor;
```

```
        return result;
    }
```

A different rounding factor is used for negative numbers as then only seven segments are available for numbers (the eighth being used for the minus).

The finished article

We now have completed JavaScript, with all the ingredients we need. The fully working calculator is shown below. Feel free to have a go at using it.



The full JavaScript for the simulation (without the HTML) is shown here, and the entire package (HTML, JavaScript and graphics) can be accessed here (87K).

The calculator as it is works fine as far as it goes. Shortcuts were taken to simplify things, and a real model would attempt to simulate all behaviour. You also notice that the code is devoid of comments. This again is for clarity, and real code would benefit enormously from liberal commenting. Finally, there are many literals in the code. JavaScript 1.1 does not allow the 'const' declaration (though later revisions do). However, it is possible to have global 'var' declarations with constant values (just be careful not to modify them---choosing all capitals for the name, say, helps to remind). Many other improvements, I'm sure, can be made and the structure of the example is not meant to be definitive in good coding.

In some senses we can say we're finished. We have a model that works. Except, how do we know it works? We'll deal with this in the next section.

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