Gerald's Column
by Gerald Fitton

I have been asked to write about some of the more obscure features of PipeDream and Fireworkz. Last month I introduced you to the Custom Function Programming Language used by both PipeDream and Fireworkz.

I'd like to make it absolutely clear that, although my write up for this article gives examples using only PipeDream, exactly the same custom function will run using Fireworkz. Indeed, any PipeDream file, including custom function files, will load into Fireworkz for RISC OS and Fireworkz for Windows. The only problems you might experience are because Fireworkz is page based and so the layout of the page may be a little different from that which you see in PipeDream. You can overcome this minor problem by loading the PipeDream file into a suitable Fireworkz template.

Good Programming

Last month I described how to call a custom function, how to pass a parameter to the function and how to return the value to the spreadsheet from which the custom function was called. This month I shall develop this theme further in three stages culminating in the technique which I prefer and which I recommend to you. Although this third technique runs rather more slowly that the first technique I use it because it complies with at least one of the major requirements of 'good programming technique', namely that it is easy to see exactly what every line does. If you can see exactly what every line in the program does then you can go back years later and extend, modify and improve the program without having to scrap what has already been written.

This Month's Example

I suggest that, at this stage in your 'lesson', you load the two files making up my example. You can find them on the Archive monthly disc or on the website. If you have a problem finding them please write to me at archive@abacusline.co.uk and I will ensure that you get a copy. These files will load into PipeDream 4, PipeDream 4.5, Fireworkz for RISC OS and Fireworkz for Windows. They will run on an A440, a RiscPC, an Iyonix and on either a native Windows machine (using Fireworkz for Windows) or within a RISC OS emulator running on a Windows machine.

Of the two files, [Add] is only 545 bytes and [c_Add] is 6284 bytes so there is not a lot to download. I suggest that you work with copies of the original files!

Loading the Files

Double click on the file [Add] and you will find that you load to the screen not only [Add] but the dependent document which I have called [c_Add] containing three custom functions. I would have liked to make [c_Add] wider but I have deliberately limited its width to 72 characters. To keep within 72 characters I have had to use short names such as "p_01" for the variables. This is not good programming practice. I would have preferred to use a long meaningful name such as "first_parameter".

Because PipeDream 4 saves the position of a PipeDream document on the screen and the position of the cursor within the document, what you should find is that [Add] is visible and alive (a yellow top bar) with the cursor in cell [Add]B7.

If this is not the case then position the pointer on the number in [Add]B7 (it should be the number 2 unless you've modified the file) and click the mouse select (left) button. You should see the same number in the formula line (at the top of the document just to the right of the cross and tick). Tap some other number, say 9, and then tap <Return>. If all goes well then, as well as [Add]B7 changing to 9, all three numbers in the range [Add]B10 to [Add]B12 increase.

Using [c_Add]add_anything(B7)

For the moment we are going to concentrate on the custom function 'called' from [Add]B10. It is not a spectacular custom function. The 'result' returned to the cell [Add]B10 is (B7 + 1), one more than the number in the cell [Add]B7.

Change the value in [Add]B7 a few times and convince yourself that the value in [Add]B10 is always 1 more than the value in [Add]B7.

Now let's have a look at the content of cell [Add]B10. Place the pointer over [Add]B10 and click select. The formula line does not show the simple, non custom function way of adding 1 which would be (B7 + 1); instead it shows the formula used to 'call' the custom function, namely [c_Add]add_anything(B7).

The 'calling' formula is in three parts:

The first part - [c_Add] - is the name of the dependent document which contains the custom function.

The second part - add_anything - is the name of the custom function within [c_Add] which you can find in row 22 of the document [c_Add].

The third part - (B7) - is the cell reference of the single piece of data passed to [c_Add] for processing.

The result of calling the custom function is returned in the cell used to call it, namely [Add]B10.

In this case the result returned to [Add]B10 is exactly the same value as that which you would have if, instead of [c_Add]add_anything(B7) you had used the much simpler formula (B7+1). However, we could have called a much more sophisticated custom function which, for example, returned the next prime number higher than the number in the cell B7.

Multiple Calling

You can use the same custom function in as many cells of the same document as you wish. For example the same custom function is called from [Add]B14.

If you place the cursor in [Add]B14 (which shows the number 1) you will find that the formula line (just to the right of the cross and tick) shows that the cell contains another 'call' to the same custom function, [c_Add]add_anything(0), but, instead of the parameter being a reference to a cell such as B7, it is a value, 0.

For example, if you can change the parameter in the formula line (of [Add]B14) from 0 to 9 then, when you tap <Return>, you will find that the custom function returns the 'result', 10 to the cell, [Add]B14, from which it was called. The number returned is always one greater than the number passed as a parameter to the custom function.

When testing a custom function sometimes it is revealing to 'call' the function from a 'spare' cell using not a cell reference as the parameter but a value you want to test.

We shall return to examples of this aid to 'debugging' later in this series.

The Custom Function [c_Add]add_anything(B7)

What we haven't looked at yet is the custom function itself to see how it adds one to the parameter and returns the 'result'. To do this we must bring the custom function, [c_Add]add_anything, into view.

Make sure that the cursor is 'in' cell [Add]B7 and then bring the document [c_Add] to the front but leave [Add] as the document which is 'alive' (a yellow top bar). If the part of [c_Add] which is visible is not rows 22 to 39 and columns A to C then use the vertical sliders or scroll bars to achieve this.

Check that it is [Add] and not [c_Add] which is alive and that PipeDream's 'input focus' (where the next character typed will appear) is in cell [Add]B7. After checking, you can confirm this most easily by tapping a new number such as 7 followed by <Return>. If you have done everything correctly then you will see that the values in cells [c_Add]C28 and [c_Add]C31 change to 7 and 8 respectively.

Just as it was possible to add one to the contents of [Add]B7 much more simply by entering (B7 + 1) into [Add]B10 instead of entering the custom function, so would it have been possible to have simplified the custom function itself to just two lines as we did in the previous Custom01 article. However, so that I can illustrate a few features of the custom function programming language, I have 'stretched out' the custom function to span lines 22 to 39 of [c_Add] and columns A to C.

Let's look at the commands in detail and see how they process data.

Row 22 - ...function("add_anything","p_01:number")

In this custom function the command   ...function   has two arguments.

The first argument is the name of the custom function, "add_anything"; note that the name is included in inverted commas.

The second argument is the one and only parameter passed to the function. You can pass many parameters each separated by a comma from the next one.

The name of every parameter must be enclosed in a pair of inverted commas.

If you want to pass a large number of values to a custom function then it is 'better' (more understandable) to pass a range or array. The name of the one parameter passed to [c_Add]add_anything is p_01 and its 'type' has been declared (after the colon) as a number. Because of this declaration, if you try to pass anything other than a number to this function then an error will be generated.

You do not have to declare the type of variable if, for example, sometimes you want to pass a number and, say, a string on another occasion.

Try typing Fred (with or without inverted commas into [Add]B7 and you will find that the error message "String not expected" is returned to cell [Add]B10. You will find the error message foreshortened in the body of the [Add] document. If you want to read the full error message then click the pointer in [Add]B10, then on the formula button (the italic f just to the right of the charts button) and finally run the pointer through the first menu option, Cell 'B10' - Slot value and you will see the full error message displayed. This technique is particularly useful when long error messages are generated.

Row 25 - ...set_value(B25,1)

Here is another convention which I recommend to you. I am using column B as a 'workspace' for my custom function sheets. One major use for this workspace is to 'store' the values of 'local' variables.

In this custom function I use cell [c_Add]B25 as storage for a 'local' variable. The 'local' variable is the number which will be added to the value taken by the parameter "p_01".

The custom function command, set_value(B25,1), writes the value 1 to the space allocated to the 'local' variable [c_Add]B25.

Change the command to set_value(B25,3) and the custom function will now add 3 to the number passed to it as a parameter. Remember that the custom function is not executed until you 'call' it from [Add]B7 (see above how to 'call' it).

Now here is something else you can do with this line which allows me to introduce the concept of a 'global' variable.

Change the command in [c_Add]A25 to set_value(B25,[Add]B8) and call the custom function by changing [Add]B7. The value which is added to the parameter is now the value of the 'global' variable in [Add]B8. Change the value in [Add]B8 and notice that the value 'returned' to [Add]B10 has not changed. Are you surprised? 'Call' the custom again by changing [Add]B7 and watch what happens.

Put [c_Add]A25 back to set_value(B25,1) or, if you have problems, delete the document and reload it.

Row 28 - ...@p_01 This command assigns the current value of the parameter "p_01" to the cell [c_Add]A28. I have included this so that you will see exactly how to include parameters in custom functions.

Notice that the inverted commas have 'disappeared' and that the name of the parameter is preceded by an @ sign. The @ sign converts a numeric parameter to a number. See what happens to the parameter if you remove the @; the three full stops in front of the p_01 disappear.

Row 31 - ...@p_01+B25

This custom function command assigns the value of (@p_01+B25) to cell [c_Add]A31. Experiment by changing the plus sign to a minus sign or, indeed, any mathematical function of two variables such as their product or quotient.

Rows 34 to 36 - set_value(C28,A28) etc

These three rows are totally unnecessary but, if you do have a problem understanding what is going on within a custom function (for example if you have a 'bug'), then lines like this are invaluable. A convention which I use for this purpose and which I recommend to you is to assign values to the cell in column C corresponding to the values in columns A or B.

Row 38 - ...result(A31)

The command ...result terminates the sequence of commands and returns a value to the cell from which it was called, namely [Add]B10.

The custom function - [c_Add]use_slots

Use the scroll bars to make lines [c_Add]A58 to [c_Add]A82 visible on the screen. This function is called from [Add]B11 and the 'result' which is returned is the sum of the numbers in the two cells [Add]B7 and [Add]B8. Note that the two values in these cells are passed as two parameters to the custom function.

Although the Custom Function Programming Language will allow it, it is bad practice to alter values of a parameter within a custom function; one of the most important features of a parameter is that, anywhere within the custom function you must be sure that it always has the same value. If you want to change it then use a 'local' variable using the technique which I describe below.

I know that the sum of two parameters can be done very simply with one line such as ...result(@p_01+@p_02) but I have written the short program in a way which illustrates the way of declaring 'local' variables and altering their values.

In this custom function I add one to the first parameter and subtract one from the second.

Of course the parameters could be processed in much more complex ways (for example, you could use trigonometrical functions to find the sum or product of two sine waves) and the 'result' returned to the cell in [Add] from which the custom function was called.

The two cells [c_Add]B65 & [c_Add]B66 are used as 'local' variables.

The values of the two parameters are assigned to these two 'local' variables by the set_value(destination,source) commands in cells [c_Add]A65 & [c_Add]A66.

As I indicated above, the reason why I have assigned the values of the parameters to two 'local' variables is so that the values can be changed. The commands in cells [c_Add]A69 and [c_Add]A70 do change the values of the 'local' variables.

The values of the two 'local' variables are added in cell [c_Add]A73.

The 'result' is returned by the command in cell [c_Add]A82.

The custom function - [c_Add]use_names

This custom function is called from cell [Add]B12 and uses cells [c_Add]A120 to [c_Add]A144. I have included this function as an introduction to the use of a 'name' for a 'local' variable within a custom function. Generally it is preferable to use a 'name' rather than a cell (as in [c_Add]use_slots) when a 'local' variable is needed. I recommend that you always use 'names' for 'local' variables as I am doing here rather than use cells as in the previous custom function.

In a custom function a 'local variable name' must be declared only once. It is good programming practice to declare all the local variables you are going to use at the beginning of the custom function as I have done here. The reason for locating all these declarations at the beginning is so that you will know where to look for all your local variables.

A 'name' is declared with the command set_name(name,slotref) as it is in [c_Add]A123 and [c_Add]A124. When a 'name' is declared in a custom function it is good practice to assign a cell in your workspace (I am using column B for my workspace) rather than assign a dummy value to the 'name'. In this case the cells B123 and B124 have been assigned to the 'names' "n_01" and "n_02" respectively. After declaring the 'name' of the 'local' variable you may assign a new value to that 'name' as many times as you wish using the command set_value(name,value). When you change the value of a name the value in the cell assigned to that name in the workspace will also change. This is a useful aid to debugging a faulty program.

The commands in cells A127 and A128 assign the values of the two parameters, "p_01" and "p_02" to the two names "n_01" and "n_02". Notice that when this command is executed new values of the 'names' are relayed to and stored in the two cells B123 and B124. The command in A138, set_value(C123C124,A123A124), is included to show that the values in A123 and A124 are not changed by the commands in A127, A128, A131 and A132! However, these last four commands do change the values of the two 'local variable names' and the values in the cells B123 and B124.

Contrast the use of 'names' in this custom function with the use of parameters in "use_slots".

Values such as "p_01" have to be written as @p_01 to become a value whereas the 'name' "n_01" is a value when written as n_01. The command set_name("name",slotref) requires inverted commas around the 'name' because, within the set_name(destination,source) command, "n_01" is a text string. The addition, n_01+n_02, in cell A135 does not require inverted commas around the 'names' nor do the 'names' need to be preceded by @; n_01 is the value taken by the 'name' "n_01".

Summary

A custom function is a sequence of commands which start with a ...function command and end with a ...result command. Once there is a custom function within a document then the whole document is a custom function document. Custom function documents behave differently from 'ordinary' documents - but they do not have a different file type!

Parameters can be passed to custom functions. It is good programming practice not to change their identity and value within the custom function.

Names should be used for local variables rather than cells.

Use the set_name(destination,source) command once only to assign a cell in the workspace to the local variable name. Thereafter, to change the value of the local variable, use the set_value(destination,source) command and not the set_name(destination,source) command.

The next tutorial

In the next tutorial of this series (if I receive enough encouragement to do so), I will introduce the concept of making a decision, "Conditional execution".

Communications

Please write to me at archive@abacusline.co.uk with your comments.