



AcroTeX.Net

AcroTeX PDF Blog

Processing Acrobat Forms using JavaScript

Internal Processing of a Field

Part 5: The Calculate Tab

D. P. Story

Warning: In Section 19.2 we discuss the (undocumented) folder-level JavaScript that ship with Acrobat/AR. The ones discussed have all been around since version 5, and the extended versions since version 6. While there is always a desire on the part of the Acrobat engineering team to maintain compatibility with previous versions, changes can occur. By using these functions directly you run the risk—slight as it might be—of incompatibilities with future versions of Acrobat. ⚠

Table of Contents

19 Calculate tab for the Text Field and Editable Combo Box	3
19.1 Custom Calculate Script	3
19.2 Built-in Calculate Script	5
19.3 Calculation Order	9
19.4 Using the Doc.calculate Property	11

19. Calculate tab for the Text Field and Editable Combo Box

In [AcroTeX Blog #20](#), we investigated the Action tab; in that article, the events of Mouse Enter, Mouse Up, Mouse Down, and Mouse Exit are triggered by direct user interaction with the document. In [AcroTeX Blog #21](#), we explored the chain of events—Keystroke, Validate, Calculate, and Format—that occur when a user enters data into a text field or editable combo box, and surveyed the important properties of the event object for each of these events.

Each of the events (Keystroke, Validate, Calculate, and Format) can have a script attached to it:

- Keystroke and Format event: Custom scripts are entered through the Format tab; in addition to custom script, there are several built-in scripts that ship with Acrobat/AD, these are available through the Format tab user interface. The Keystroke and Format scripting were covered in [AcroTeX Blog #22](#).
- Validate event: A custom script can be entered through the Validate tab; there is also a built-in script for validating common requirements, this script is available through the Validate tab. Validate scripting was studied in [AcroTeX Blog #23](#).
- Calculate tab: a custom script can be entered through the Calculate tab; also included in this tab are some built-in scripts for making common calculations, these scripts are available through the user interface of the Calculate tab.

In this article, we study the scripting associated with the Calculate tab. We shall discuss custom Calculate scripts, as well as survey the built-in scripts that ship with Acrobat/AD and are available through the user interface.

Acknowledgements: I extracted some of the information found in Section 19.2 by snooping through a PDF file using the [PDF CanOpener](#), by [WindJack Solutions, Inc.](#)^{1, 2}

Section 2.3.6.1 of *The TeX Web Companion, Integrating TeX, HTML, and XML*, by Michel Goossens, Sebastian Rahtz, *et al*, was also used as a resource for Sections 19.2. This book is one of the few that documents (at least partially) the folder level JavaScript functions that are used by the built-in scripts of the Format, Validate, and Calculate tabs.

19.1. Custom Calculate Script

A *calculated field* is one whose value is calculated from the values of other fields; that is, one that has a Calculation script. Text is not usually entered directly into a calculated field through direct keystroke input.

¹<http://www.windjack.com>

²I downloaded a fully functional version of the product, an Acrobat plug-in, for a 10 day trial period. I found the [PDF CanOpener](#) to be intuitive and very easy to use. It is indeed an excellent tool for examining (and editing) the structure of a PDF file.


```

1 var start=this.getField("odStart").value;
2 var end=this.getField("odEnd").value;
3 if ( start==" || end==" ) event.rc=false;
4 else event.value=Math.abs(end-start);

```

Code Comments: We get the start and end values in lines (1) and (2). If either values is the empty string, in line (3), we do nothing (`event.rc=false`); otherwise we take the difference in start and end, and take the absolute value in case the user has his mileage reversed. We then set `event.value` to this calculated value in line (4).

The Calculate script for the reimbursement field follows:

```

1 var miles=this.getField("totalMiles").value;
2 if ( miles==" ) event.rc=false;
3 else event.value=0.27*miles;

```

Code Comments: This is a more routine example of a custom calculation. We get the value of miles, if it is the empty string we set `event.rc` to false; otherwise, we set `event.value` to `0.27*miles`. Line (2) is optional, without it, when the fields are reset, the reimbursement field will have a value of \$0.00. I prefer the field to be blank when there is no value. □

Another custom script is given in [Example 19.4](#) on page 7.

19.2. Built-in Calculate Script

The Calculate tab, in addition to the custom calculation script editor, also offers a built-in calculation feature, and access to editor for using a simplified field notation. Through the built-in calculation feature, you can sum, multiply, and average the values of specified fields, to the calculate the minimum and maximum values of the specified fields.

The built-in calculation feature is implemented with the function `AFSimple_Calculate`, which is defined at the folder level. The parameters of this function are listed below.

```
AFSimple_Calculate(cFunction, cFields)
```

Description of Parameters:

- `cFunction` is a string that identifies the operation to be performed on the fields, recognizable values of `cFunction` are SUM, PRD, AVG, MIN, MAX.
- `cFields` is an array of the names of the fields used to calculate the value of this field; for example, `new Array("Text1", "Text2")`.

Example 19.2. Illustrating the use of `AFSimple_Calculate`. Enter numbers in the fields in the left column, the sum, product, average, minimum, and maximum are computed in the right column.

Numerical Fields	Operation	Calculated Fields
	SUM	
	PRD	
	AVG	
	MIN	
	MAX	

The Calculate scripts for each of the field in the right column are the same, except for the specification for `cFunction` parameter, below is the one for the SUM operation:

```
AFSimple_Calculate("SUM", new Array("txt19-2-1", "txt19-2-2",
    "txt19-2-3", "txt19-2-4", "txt19-2-5"))
```




Let us repeat the same example, this time using a hierarchical naming scheme of the fields in the left column. This makes them easier to reference in the `cFields` parameter of the built-in function `AFSimple_Calculate`.

Example 19.3. AFSimple_Calculate with a hierarchical naming scheme. Enter numbers in the fields in the left column, the sum, product, average, minimum, and maximum are computed in the right column.

Numerical Fields	Operation	Calculated Fields
	SUM	
	PRD	
	AVG	
	MIN	
	MAX	

The parent name of each of the fields in the left column is "txt19-3", the fields are named "txt19-3.1"-"txt19-3.5". With this hierarchical naming scheme, it suffices to reference these five field using the parent's name. Below is the Calculate script for the sum of the fields, `cFields` is very simple.

```
AFSimple_Calculate("SUM", new Array("txt19-3"))
```

The ability to reference the parent's name is also available through the user interface. 

One of the big complaints about the built-in function `AFSimple_Calculate` is that it does not bring back the expected results for the operation of AVG (averaging). For example, when

only the first field has a value of 5, and the others are empty, the average of these five fields is computed to be 1 ($SUM/n = 5/5 = 1$). Try this on any of the above examples. For a spreadsheet application, when 4 of the 5 cells are empty, the average function would return 5. Let's try to create a custom averaging function.

Example 19.4. Averaging à la spreadsheet. We write our own averaging function, defined at the document level. For your convenience, the verbatim listing follows.

In it, we use the built-in function `AFSimple_Calculate` to calculate the sum of the fields, but we need to be careful about counting the number of fields. We count only the ones that have a value (are non-empty), line (7). We also protect ourselves against division by zero in line (9), if all fields are empty, we give the calculated field an empty string as its value.

```

1  function myAVG_Calculate(aFields) {
2      AFSimple_Calculate("SUM", aFields);
3      var f,a,n=0;
4      for (var i=0; i<aFields.length; i++) {
5          f=this.getField(aFields[i]);
6          a=f.getArray();
7          for (var j=0; j<a.length; j++) if (a[j].valueAsString!="") n++;
8      }
9      if ( n==0 ) event.value="";
10     else event.value /= n;
11 }

```

In line (7), we use the Field property `Field.valueAsString`, this property enables us to determine if there is an entry in the field. If we had used `Field.value`, the test `(a[j].value!="")` is false if the field when the entry in the field is the empty string or if the entry is the number 0; consequently, when there is a 0 in a field, the variable `n` *does not get incremented* as it should be. (In a spreadsheet application, if a cell has a 0 in it, then it is averaged in; if a cell is empty, that cell is not averaged in.)

Numerical Fields	Operation	Calculated Fields
	SUM	
	PRD	
	AVG	
	MIN	
	MAX	

The Calculate script for the calculated field in the AVG row is

```
myAVG_Calculate(new Array("txt19-4-1","txt19-4-2","txt19-4-3",
    "txt19-4-4","txt19-4-5"))
```

The use of a hierarchical naming scheme for the fields in the left column would have simplified the specification of the above array. □

Example 19.5. Clearing a Calculation Field on Reset. One thing that has bothered me about calculated fields is that they don't reset to an empty value. The left column of the example below, represents the "default" behavior of a calculated field. However, by using a Validate script, you can force the calculation field to be empty on reset, see the same set of fields in the right column in the example below.

Enter a number in the two top rows, the third row is the sum of the two rows above it. Reset after entering your numbers, and observe the values of the calculated fields.

The code for the Validate script for the last field in the right column is

```
clearCalcFieldonReset("txtNum1-wReset","txtNum2-wReset");
```

The function `clearCalcFieldonReset` is defined in the document JavaScript section of the document, it is listed under the name **Clear Calculate Field on Reset**. The verbatim listing follows:

```
1 function clearCalcFieldonReset() {
2     for ( var bReset=true, i=0; i<arguments.length; i++){
3         var f=this.getField(arguments[i]);
4         var a=f.getArray();
5         for (var j=0; j<a.length; j++)
6             if (a[j]!=null&&a[j].value!="") {bReset=false; break};
7         if (!bReset) break;
8     }
9     if (bReset) event.value="";
10 }
```

The parameters for `clearCalcFieldonReset` is a list of the field names that are used to calculate the value of the field, in this case, they are `"txtNum1-wReset"` and `"txtNum2-wReset"`. The function gets the field names from the function arguments, and loops through to see if any of these fields is non-empty; if all are empty, we set `event.value=""`. Here, `event.value` is the value of the field executing the Validate script. □

19.3. Calculation Order

The order of calculation can be important. Acrobat/AR maintains an array of all calculated fields. The order the fields are calculated follows the entries in this *calculation array*. For a document that has a calculated field whose value depends on another calculated field, the first field must be calculated before the second one is. The document author, through the user interface, may have to re-arranged the order of calculation to be assured that all calculated fields are updated correctly. [Example 19.7](#), page 10, tries to illustrate this point. The Field property `Field.calcOrderIndex` can be used to programmatically change the order of calculation.

Below is a task-oriented example, it shows how to get a list of all calculated fields.

Example 19.6. Acquiring a list of all calculated fields.

This script lists all fields that are calculated. We go through all fields using Doc method `Doc.getNthFieldName`; we get the `Field.calcOrderIndex` value for each field. If the value is -1, that field is not a calculated field; otherwise, it is.

```
for (var i=0; i<this.numFields; i++) {
    var fname=this.getNthFieldName(i);
    var f=this.getField(fname);
    if ( f.calcOrderIndex !=-1 )
        console.println("Field ["+fname+"] has calc index: "
            +f.calcOrderIndex);
}
```

Get all calculated fields in the [console window](#):

A variation—one of many, depending on how you want the information—get all calculated fields, and load them into a matrix sorted by index. Print out the calculated fields according to the order of calculation, results written to the [console window](#):

```
var aCalcFields=new Array();
for (var i=0; i<this.numFields; i++) {
    var fname=this.getNthFieldName(i);
    f=this.getField(fname);
    if ( f.calcOrderIndex !=-1 )
        aCalcFields.push([f.calcOrderIndex,fname]);
}
var aCalcFields=aCalcFields.sort(function(a,b){return a[0]-b[0];});
for (var i=0; i<aCalcFields.length; i++)
    console.println("calcOrderArray["+i+"]="+aCalcFields[i][1]);
```

We now present an example of programmatically changing the order of calculation. □

Example 19.7. Reordering Calculation Order. This is the same example as Example 19.5, page 8; however, the reimbursement field has been move up in position on the page. The authoring system that I use, called \LaTeX , will create the reimbursement field before it creates the total mileage field. This causes problem since the value of the reimbursement field depends on that of the total mileage field.

Enter values for the start and end odometer readings (or enter total mileage directly); you'll see that the reimbursement value does not get updated. That's because it gets calculated before the value of the total mileage field is updated. (Reimbursement calculated at a rate of \$0.27 per mile.) Change the values, now the reimbursement field has a value, but it is the wrong value; it is the one based on the previous values.

Now, click the Reorder button, this button re-arranges the order of calculation, and all is well in the world again.

1. Enter the starting and ending odometer values in the two text field, so labeled. The mileage will be calculated automatically in the third box labeled Total mileage.
2. Enter your total mileage directly into the text field labeled total mileage, leaving the other two field with no value.

<u>Mileage</u>	<u>Travel Reimbursement Due</u>
Start Odometer:	
End Odometer:	
Total Mileage:	

The code for the Reorder button is a MouseUp action:

```

1 var miles=this.getField("totalMiles-CalcOrder");
2 var reimburse=this.getField("reImbursement-CalcOrder");
3 if (reimburse.calcOrderIndex < miles.calcOrderIndex )
4     reimburse.calcOrderIndex=miles.calcOrderIndex+1;
5 var milesValue=miles.value;
6 this.resetForm(["totalMiles-CalcOrder"]);
7 this.getField("totalMiles-CalcOrder").value=milesValue;
```

Code Comments: Get the Field objects for the two fields in question (lines (1) and (2)). If the calculate index of `reimburse` is less that that of `miles`, we make an adjustment to the order, lines (3) and (4). Lines (5)–(7) are optional; we save the value of the `miles` field, we reset the `miles` field, then re-populate the `miles` field with that value. (Resetting and re-populating forces the calculate chain of events to occur; hence, updating the `reimburse` field.) □

19.4. Using the `Doc.calculate` Property

When a document contains a large number of calculated fields, each time a form field is changed, the whole chain of Calculate events occurs to update any of the calculated fields; this chain of events even occurs when none of the calculated fields depends on the value of the field changed. Generally, a large number of calculated fields can slow down processing quite a bit, the user can get frustrated when there is such a delay even when is not needed. There is a Boolean property `Doc.calculate`, when `true` calculations may be performed on the document; when `false`, calculation cannot be performed. Judicious placement of `this.calculate=false` and `this.calculate=true` can be of great value to the user, and improve is form filling experience.

To illustrate, consider the two fields below. The one on the left is a text field that does nothing other than to accept input. When you enter text and commit it, you'll see the calculator cursor, this indicates the Calculate chain of events are occurring and all calculated fields are being updated (even though this field does nothing). The beautiful field on the right, however has `this.calculate=false` as part of its Focus script and `this.calculate=true` as part of its Blur script. Enter text into the field on the right, and observe the cursor as you commit the data. See anything?

Did you have a better experience with the field on the right? I'm sure you did.

Another suggested use of `Doc.calculate` is in a reset button. Consider the following two reset buttons (1) (2) Again, the first button executes with the calculator icon appear, while the second one executes without the calculator icon appearing. Compare the results of the two reset buttons on the last field in each of the two columns in [Example 19.5](#), page 8.

The second button has `MouseUp` action,

```
this.calculate=false;
this.resetForm();
this.calculate=true;
```

Well, that's pretty much it for now, I simply must get back to my retirement. ☹