



AcroTeX.Net

## AcroTeX PDF Blog

# Creating Acrobat Form Fields using JavaScript

D. P. Story

The source file for this [AcroTeX PDF Blog](#) is the batch sequence [Insert Navigation Buttons](#), described in [Section 1.4](#), page 13.

## Table of Contents

<b>1</b>	<b>Creating Acrobat Form Fields using JavaScript</b>	<b>3</b>
1.1	Doc.addField and Field.setAction . . . . .	3
1.2	Creating Fields and Actions by User Interaction . . . . .	5
	• Text Fields . . . . .	5
	• Push Buttons . . . . .	7
	• Check Boxes . . . . .	8
	• Radio Buttons . . . . .	9
	• List and Combo Boxes . . . . .	11
1.3	Creating Fields using a Batch Sequence . . . . .	12
1.4	Executing a script through an FDF . . . . .	13

## 1. Creating Acrobat Form Fields using JavaScript

The dynamic creation of form fields is somewhat limited, in my opinion. The two methods for creating Acrobat forms and actions are `Doc.addField` and `Field.setAction`. From the quick bar of the *JavaScript for Acrobat API Reference*, we see that the former method requires Form Reader Rights (it will work in Acrobat, but not in AR unless the additional Reader rights are available), the latter is never available in AR (but, again, is available in Acrobat).

I consider these two methods part of *document development*. The document author, or developer, who uses some flavor of Acrobat (Pro/Pro Extended), might use these methods in the process of constructing a document to be distributed.

There are three steps to create a form field:

1. Create the field using `Doc.addField`, and acquire the returned Field object.
2. Set the properties of the field using the Field object from step 1.
3. Set any JavaScript actions desired using `Field.setAction`.

Naturally enough, we can set properties and actions in any order we wish.

### 1.1. Doc.addField and Field.setAction

The syntax for `Doc.addField` is

```
var oField=this.addField({
  cName: string,
  cFieldType: text | button | combobox | listbox |
             checkbox | radiobutton | signature,
  nPageNum: number,
  oCoords: [ x_ul, y_ul, x_lr, y_lr ]
})
```

where,

- `cName` is a string that is used for the name of the field.
- `cFieldType` is a string that defines the field type that is to be created.
- `nPageNum` is the page number the field is to be created on. The page number is 0-based,
- `oCoords` is an array of coordinates that define the bounding rectangle of the field. These coordinates are given in *Rotated User Space*.<sup>1</sup> In rotated user space, the origin is always the lower left corner of the crop box.

<sup>1</sup>For an explanation of Rotated User Space, see the article *Automating Placement of Annotations*, by Thom Parker, at [AcrobatUser.com](http://AcrobatUser.com), the Adobe Acrobat User Community.

`Doc.addField` returns the `Field` object of the newly created field. The returned object should be assigned to a JavaScript variable, `oField`, in the above specification.

Because we must specify the coordinates of the bounding rectangle, placing a field becomes a problem with this method. There are two strategies I have used to set the location of a field created using `Doc.addField`:

1. Place the field or fields in a fixed location, perhaps relative to one of the page corners, in a space that is known to be white space (no content in that region).
2. Place the field or fields in a location relative to another field that is already there.

Unless you are 100% sure the end-user will be using Acrobat, the next question is when do you use `Doc.addField`? I propose the following:

1. You can create fields dynamically by having the user interact with a document (pushing a button, or entering text in a text field); however, this method requires the end user to have Acrobat.
2. `Doc.addField` in a batch sequence
3. Execute `Doc.addField` in the JavaScript Debugger window
4. Place the script in an FDF that contains the JavaScript that will be executed when it is imported. This is the method I use, and is the one most convenient to my own workflow. My workflow is to create content using the markup language  $\LaTeX$ , convert the natural output format (DVI) to PostScript, and finally, distill the PostScript. When the distiller has finished generating the PDF, and the file is opened in Acrobat for the first time, the FDF (filled with JavaScript) is imported and executed. Both content and JavaScript are written in the  $\LaTeX$  source file.

In this article, I'll attempt to illustrate these four methods, well, maybe I won't bother with (3).

To assign JavaScript actions to a field—even a field not necessarily created by `Doc.addField`—use `Field.setAction`. The parameters for this method are given below.

```
Field.setAction({
  cTrigger: MouseUp | MouseDown | MouseEnter | MouseExit |
            OnFocus | OnBlur | Keystroke | Validate | Calculate | Format,
  cScript: string
})
```

The triggers are the same ones discussed in [AcroTeX Blog #20](#) (mouse, focus, blur triggers discussed); [AcroTeX Blog #21](#) (discusses the event chain); [AcroTeX Blog #22](#) (keystroke and format discussed); [AcroTeX Blog #23](#) (validate discussed); and [AcroTeX Blog #24](#) (calculate discussed).

The parameter `cScript` is used to define the JavaScript action, because `cScript` is a string, this leads to problems in creating long or complex scripts. It's best to simply call a JavaScript function already defined at the document level.

**Limitations:** `Field.setAction` cannot get an action, it can only set an action. *It will overwrite an already existent action*; consequently, you cannot append code to existent code.

## 1.2. Creating Fields and Actions by User Interaction

This is the case that is of limited value in that the user needs to have Acrobat, but it is the one that is most fun!

In the following examples, we draw liberally on the techniques illustrated in AcroTeX Blog #12–AcroTeX Blog #24. For the examples in this section to work, the reader, that you, must not have Reader, that's not you. The examples will work for Acrobat users.

### • Text Fields

We create a text field, and a multiline text field in the next two examples.

**Example 1.1. Creating a text field** Position the text field relative to the location of the button.

Below is the verbatim listing of the push button action.

```
1  var g=this.getField("txt1-1-1");
2  if ( g==null ) {
3      var f=event.target;
4      var myRect=f.rect;
5      var deltaX=6;
6      var widthTxt=72*1.5;
7      myRect[0]=myRect[2]+deltaX;
8      myRect[2]=myRect[0]+widthTxt;
9      var f=this.addField("txt1-1-1", "text", this.pageNum, myRect);
10     f.delay=true;
11         f.strokeColor=color.black;
12         f.textColor=color.green;
13     f.delay=false;
14 }
```

**Code Comments:** First we see if the field has already been created (lines (1) and (2)), if no, we create it. We get the Field object of the button (3), and its bounding rectangle (4). We'll move over 6 points horizontally (5), and set the field width (6). We'll make the field the same height as the button. In lines (7) and (8) we calculate the new x-coordinates. Finally, we create

the text field using `Doc.addField`, and the array we manipulated, `myRect`, as the rectangle argument. We begin setting properties delaying the redrawing of the field (10), we set our desired properties (11)–(12), then allow the field to be redrawn, line (13).

Note that the parameters for `addField` are entered in traditional functional form; alternatively, we could have used the object literal method of specifying the parameters, like so,

```

1 var f=this.addField({
2     cName: "txt1-1-1", cFieldType: "text",
3     nPageNum: this.pageNum, oCoords: myRect
4 });

```

The advantage with this method is that the parameters can be listed in any order, and if there are optional parameters, they can be included or not in the object. □

A field can be removed using the `Doc.removeField` method, by passing the field name as an argument. Let's remove the field just created above.

### Example 1.2. Remove a field using `Field.removeField`

The MouseUp code for this button is `this.removeField("txt1-1-1");` □

**Example 1.3. Creating a multiline Text Field.** Position the text field relative to the location of the button.

Below is the verbatim listing of the push button action.

```

1 var g=this.getField("txt1-3-1");
2 if ( g==null ) {
3     var f=event.target;
4     var myRect=f.rect;
5     var deltaX=6;
6     var widthTxt=72*3;
7     var heightTxt=5*12;
8     myRect[0]=myRect[2]+deltaX;
9     myRect[2]=myRect[0]+widthTxt;
10    myRect[1]=myRect[3]+heightTxt;
11    var f=this.addField("txt1-3-1", "text", this.pageNum, myRect);
12    f.delay=true;
13        f.multiline=true;
14        f.strokeColor=color.red;
15        f.value="Ooops! I messed up! AcroTeX is on the Rocks!";
16    f.delay=false;
17 } else this.removeField("txt1-3-1");

```

**Code Comments:** This example illustrates the dangers of dynamically creating fields. You must have a detailed knowledge of the surrounding content. I can correct this error by subtracting a fixed amount from the y-coordinates. This code is the same as the [Example 1.1](#), page 5, the differences in line (8), the introduction of a height; line (11), the calculation of the y-coordinates of the text field; line (14), setting the *Field*.multiline property to true; and line (16), giving the field a funny value. □

There are few examples where a dynamic creation of a field through user interaction. Indeed, all this can be accomplished by using static fields (created by the user interface) and making them hidden. There may be several fields overlaying each other, the one you need can be made visible, the then made hidden again.

### • Push Buttons

Push buttons were fully discussed in [AcroTeX Blog #14](#).

The *Field.setAction* method has not been illustrated yet. The push button is an ideal choice for creating actions.

**Example 1.4. Creating a Push Button with Action.** Place the button in the bottom margin of this page, centered horizontally.

Below is the verbatim listing of the push button action. We assume there is a one inch margin at the bottom of the page to insert the button in. We use *Doc.getPageBox* to get the dimensions of the crop box, the one used by all Acrobat fields.

```
1  var g=this.getField("pb1-4-1");
2  if ( g==null ) {
3      var aRect=this.getPageBox("Crop", this.pageNum);r
4      var widthPage=aRect[2]-aRect[0];
5      var xcoordCenter=widthPage/2;
6      var widthBtn=32;
7      var myRect= new Array();
8      myRect[0]=xcoordCenter-(widthBtn)/2;
9      myRect[2]=xcoordCenter+(widthBtn)/2;
10     myRect[3]=72*.5;
11     myRect[1]=myRect[3]+11;
12     var f=this.addField("pb1-4-1", "button", this.pageNum, myRect);
13     f.delay=true;
14         f.strokeColor=color.black;
15         f.textColor=color.red;
16         f.fillColor=color.ltGray
17         f.highlight=highlight.p;
```

```

18         f.borderStyle=border.b;
19         f.textSize=0;
20         f.buttonSetCaption("Push", 0);
21         f.buttonSetCaption("Me", 2);
22         f.buttonSetCaption("Now!", 1);
23         f.delay=false;
24         var cMsg="Hello World! Have I ever told you that \\\"AcroTeX Rocks!\\\"?"
25         f.setAction("MouseUp", "app.alert(cMsg)");
26     } else this.removeField("pb1-4-1");

```



## • Check Boxes

We create a check box, first discussed in [AcroTeX Blog #15](#).

### Example 1.5. Creating a check box using addField.

#### Comments:

- The first two check boxes have the same names, they are widgets with terminal name cb1-5-1. The export values of the widgets are different. You'll note they behave like radio buttons, except you can uncheck the field. When you query the field, the value of the field is one of three values, Yes1, Yes2, or Off.
- The second two fields have the same name (hence, are widgets with terminal name cb1-5-2) and the same export values; these two work in tandem.
- The last two have different names—a standard setup—and, therefore, independently of each other (and from the first four check boxes).

The verbatim listing of the push button action is

```

1  var g=this.getField("cb1-5-1");
2  if ( g==null ) {
3      var myCBs= [
4          ["cb1-5-1", ["Yes1", "Yes2"]], ["cb1-5-1", ["Yes1", "Yes2"]],
5          ["cb1-5-2", ["Yes1", "Yes1"]], ["cb1-5-2", ["Yes1", "Yes1"]],
6          ["cb1-5-3", ["Yes1"]], ["cb1-5-4", ["Yes2"]]
7      ];
8      var f=event.target;
9      var myRect=f.rect;
10     var deltaX=6;
11     var widthCB=13;
12     var x_l=myRect[2]+deltaX;
13     for (var i=0; i<myCBs.length; i++) {
14         myRect[0]=x_l;

```

```

15     myRect[2]=myRect[0]+widthCB;
16     x_l=myRect[2]+deltaX;
17     var f=this.addField(myCBs[i][0], "checkbox", this.pageNum, myRect);
18     f.delay=true;
19         f.exportValues=myCBs[i][1];
20         f.strokeColor=color.black;
21         f.textColor=color.red;
22     f.delay=false;
23 }
24 } else for (var i=0; i<myCBs.length; i++) this.removeField(myCBs[i][0]);

```

**Question:** How do you attach actions to *each* of these check boxes. The first two are widgets, the second two are widgets, the third two are fields with different names. [Click here](#) for the answer, but only after you have meditated on the solution.

**Note:** An action can be removed by overwriting the current action; if you pass the empty string as script, you remove the action, and replace it with no action, for example,

```
Field.setAction("MouseUp", "");
```

removes the current MouseUp action of the referenced field.

### • Radio Buttons

When creating a set of radio buttons belonging to the same group, use the same name for each. Radio buttons also have a property *Field.radiosInUnison*: If *false* (the default), if a set of radio buttons have the same name and export value, they behave in a mutually exclusive fashion, like HTML radio buttons; if *true*, a set of radio buttons with same name and export value turn on and off in unison (this was the behavior seen in Acrobat 4.0).

In the example below, we illustrate radio button fields, with *Field.radiosInUnison* set to *false*, then to *true*.

#### **Example 1.6. Creating Radio Buttons.**

##### **Field.radiosInUnison set to false.**

The first two radio buttons has the same export value (Yes1), yet the field behaves in a mutually exclusion fashion.

The verbatim code for the push button is

```

1 var rbfGroupName="rbf1-6";
2 var g=this.getField(rbfGroupName);
3 if ( g==null ) {

```

```

4     var aEVs= ["Yes1","Yes1","Yes2","Yes3","Yes4"];
5     var f=event.target, myRect=f.rect;
6     var deltaX=6, widthRBF=13;
7     var x_l=myRect[2]+deltaX;
8     myRect[0]=x_l;
9     myRect[2]=myRect[0]+widthRBF;
10    var rbf=this.addField(rbfGroupName,"radiobutton",this.pageNum,myRect);
11    for (var i=1; i<aEVs.length; i++) {
12        x_l=myRect[2]+deltaX;
13        myRect[0]=x_l;
14        myRect[2]=myRect[0]+widthRBF;
15        this.addField(rbfGroupName,"radiobutton",this.pageNum,myRect);
16    }
17    f.delay=true;
18        rbf.strokeColor=color.black;
19        rbf.textColor=color.red;
20    f.delay=false;
21    rbf.exportValues=aEVs;
22 } else this.removeField(rbfGroupName);

```

**Code Comments:** Note that we acquire the field object of the radio button field in line (12), while we don't get the field object in line (17). (Why?) In lines (20) and (21), we set properties of the radio buttons. These properties are widget properties, but the object is a field object for the whole group; so setting them with a field object representing the group, sets all widgets (review table presented in the introduction to the [Field object](#) in the *JavaScript for Acrobat API Reference*. Finally, in line (23) we set the export values of the buttons in the group.

### Field.radiosInUnison set to true.

The first two radio buttons has the same export value (Yes1), yet the field behaves in a mutually exclusion fashion. Note the difference in the behavior of the first two buttons in this field, and the previous radio button field. Setting *Field.radiosInUnison* to *true* when you want to transfer results of a radio button field from one page to another, for example.

The verbatim code for the push button is same as above, with a few exceptions, highlighted below.

```

1     ...
2     var rbf=this.addField(rbfGroupName,"radiobutton",this.pageNum,myRect);
3     ...
4     rbf.radiosInUnison=true;
5     rbf.exportValues=aEVs;
6 } else this.removeField(rbfGroupName);

```

**Code Comments:** The new line is highlighted in red, we set *Field.radiosInUnison* to *true*. □

## • List and Combo Boxes

### Example 1.7. Creating a list box and combo box.

What is your favorite piece of clothing? Use a list box or a combo box to make the choice to make your choice. When you are finished choosing your favorite item, you may **reset** the fields to let someone else have a chance to make a choice.

The MouseUp action for the radio button fields calls the function `createLBRBF`, which is implemented as document JavaScript. The function `createLBRBF` is listed below:

```
1 function createLBRBF(hiddenField) {
2     var f=this.getField("myApparelChoice");
3     if ( f!=null) this.removeField("myApparelChoice");
4     var fieldtype=event.target.value;
5     var btn=this.getField(hiddenField);
6     var myRect=btn.rect;
7     var width=72;
8     var height=(fieldtype=="listbox") ? 5*13 : 13;
9     myRect[2]=myRect[0]+width;
10    myRect[3]=myRect[1]-height;
11    var f=this.addField("myApparelChoice",fieldtype,this.pageNum,myRect);
12    f.setItems([[["Socks","1"],["Shoes","2"],["Pants","3"],
13                ["Shirt","4"],["Tie","5"]]);
14    f.setAction("Keystroke", "lbrbfKeystroke()");
15    f.delay=true;
16        f.defaultValue="1"; f.value="1";
17        f.strokeColor=["RGB",0,0.6,0];
18        f.fillColor=["RGB",0.98,0.92,0.73];
19    f.delay=false;
20 }
```

I've created a hidden push button (using the `pdfmark` operator, which is what my authoring system, the [AcroTeX eDucation Bundle](#), uses. The field name of this hidden button is passed as a parameter, this was necessary since I use an automated naming convention, based on the example number. In line (8), the `height` is set depending on whether we are creating a list box or a combo box. We insert the choices in line (12) and (13); and set the keystroke script in line (14). The keystroke script is defined at the document level as well, see script named **List Box and Combo Box JS** at the document level. □

### 1.3. Creating Fields using a Batch Sequence

The batch sequence is considered a trusted platform—along with the JavaScript Debugger Console window—on which all JavaScript methods can execute safely, including the security restricted methods. The theory is that the user should know what the batch sequence he executes does, and in the case of an **Execute JavaScript** batch sequence, he should know what the code does and whether it is safe to execute on his computer.

For a batch sequence, you are somewhat blind, so you must place fields in known what spaces, such as the margins of a document, assuming the document has a known white space in the margins.

**Insert Navigation Buttons.** That having been said, I've attached a modified version of a batch sequence I wrote several years ago. The name of the batch sequence is **Insert Navigation Buttons**.<sup>2</sup> The batch creates four buttons at the bottom of each page of any document that is selected for the batch. The buttons are *Next Page*, *Prev Page*, *Next View*, *Prev View*.<sup>3</sup> Once you analyze the batch sequence and understand it, you are at liberty to change it as desired.

**Installation.** If you are unfamiliar with how to install a batch sequence, here is how it is done.

- Save the file **Insert Navigation Buttons.sequ** to your hard drive.
- Open the JavaScript Debugger Console Window (Ctrl+J). Enter the code `app.getPath({cCategory: "user", cFolder: "sequences"})` or [click this link](#) for a copy of the code to appear in the console window.
- Execute the code in the console window, by placing the cursor on the line containing the code and pressing the Ctrl+Enter key, or Enter on the keypad. The method returns the path to the Sequences folder, this is where the file `Insert Navigation Buttons.sequ` needs to be placed. Place it there.
- Close Acrobat, if open, then start Acrobat again. The batch sequence **Insert Navigation Buttons** should be listed under `Advanced > Document Processing > Batch Processing`

The batch is initially set up to process any file open in Acrobat, but this can be changed. Open this file, if you wish, in Acrobat and take the batch for a test spin.

<sup>2</sup>The file name is `Insert Navigation Buttons.sequ`.

<sup>3</sup>The `Next View` and `Prev View` use the methods `app.goBack()` and `app.goForward()`, respectively. In testing the batch, I've discovered these two methods don't work in Acrobat 9.1; I've since discovered these methods do not work for version 8 or version 9. This may be additional security, or a bug. I have "a call in" to the Acrobat Engineering Team to resolve this problem.

## 1.4. Executing a script through an FDF

Another approach that I take, especially in my  $\text{\LaTeX}$  workflow is to import an FDF containing document level JavaScript and/or script that is meant to be executed once, then discarded. Below is an example, which does not illustrate the use of `Field.addField`, but does give you the idea of usage.

```

1  %FDF-1.2
2  1 0 obj
3  << /FDF << /JavaScript << /Doc 2 0 R /After 3 0 R >> >> >>
4  endobj
5  2 0 obj
6  [ (ExecJS docassembly) (var _docassembly = true;) ]
7  endobj
8  3 0 obj
9  <<>>
10 stream
11 aebTrustedFunctions(this, aebAddWatermarkFromFile, {
12     bOnTop:false,
13     cDIPath:"/C/AcroTeX/AcroTeX/ManualBGs/PDF_Blog_BG.pdf"
14 });
15 endstream
16 endobj
17 trailer
18 << /Root 1 0 R >>
19 %%EOF

```

I also have an open page action that gets automatically created, the code I use looks like this

```

1  if(typeof _docassembly == "undefined")
2      ( app.viewerVersion > 8 ) ?
3      aebTrustedFunctions( this, aebImportAnFDF, "docassembly.fdf" ) :
4      this.importAnFDF("docassembly.fdf");

```

When the document is opened for the first time after distillation, the open page action, above, imports the FDF into the document provided the variable `_docassembly` is undefined (this prevents the script from being imported every time the document is opened).

The functions that are called, both in the open page action and in the FDF itself use security restricted functions, so at the folder level, I have a function `aebTrustedFunctions` that calls these security restricted functions as trusted function.

By the way, this code is the one I use to add the border onto the [AcroTeX PDF Blogs](#) using the `Doc.addWatermarkFromFile` method. Sweet system.

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

### Answer to the question posed in Example 1.5

For the first two, which are widgets with the same terminal name, do you remember how to reference widgets?

```
1 var f=this.getField("cb1-5-1.0");
2 f.setAction("MouseUp", "app.alert(\\\"First widget action\\\")");
3 var f=this.getField("cb1-5-1.1");
4 f.setAction("MouseUp", "app.alert(\\\"Second widget action\\\")");
```

The second two, also widgets with the same parent name, are done similarly.

The last two are “standard” check box fields, the, for example,

```
1 var f=this.getField("cb1-5-3");
2 var cMsgi="Action of the field \\\"cb1-5-3\\\"";
3 f.setAction("MouseUp", "app.alert(cMsgi)");
4 var f=this.getField("cb1-5-4");
5 var cMsgii="Action of the field \\\"cb1-5-4\\\"";
6 f.setAction("MouseUp", "app.alert(cMsgii)");
```

Is that clear?

Return to [Example 1.5](#), page 8, and “check” it out. ☞

