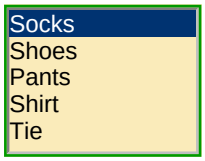
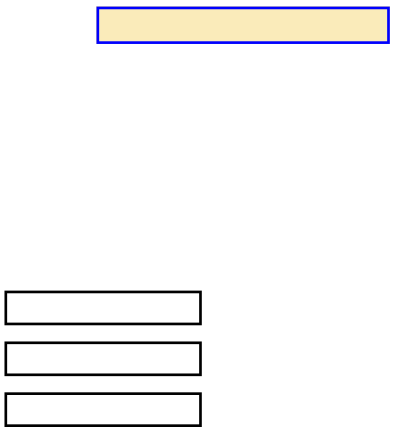

Push


Push


Socks
Shoes
Pants
Shirt
Tie


Socks
Get



3. **Mouse Down:** The event is triggered when the (left) mouse button is pushed down while the mouse is within the bounding rectangle of the field. The \AAMouseDown key is used within the argument of \AA to define a mouse down event:

```
\pushButton[\AA{\AAMouseDown{\JS{app.alert("Mouse Down!")}}}]
    {myButton}{30bp}{12bp}
```

4. **Mouse Up:** The event is triggered when the (left) mouse button is released while the mouse is within the bounding rectangle of the field. The \A key (or \AAMouseUp key is used within the argument of \AA) is used to define a mouse up event:

```
\pushButton[\A{\JS{app.alert("Mouse Up!")}}]
    {myButton}{30bp}{12bp}
```

The same code can be performed as follows:

```
\pushButton[\AA{\AAMouseUp{\JS{app.alert("Mouse Up!")}}}]
    {myButton}{30bp}{12bp}
```

When both types of mouse up actions are defined for the same field, the one defined by \A is the one that is executed.

5. **On Focus:** The event is triggered when the field comes into focus (either by tabbing from another field, or clicking the mouse within the bounding rectangle. The \AAOnFocus key is used within the argument of \AA to define an 'on focus' event:

```
\textField[\AA{\AAOnFocus{\JS{%
    app.alert("Please enter some data!")}}}]
{myText}{1.5in}{12bp}
```

6. **On Blur:** The event is triggered when the field loses focus (either by tabbing to another field, by clicking somewhere outside the field, or (in the case of a text field, for example) pressing the Enter button. The \AAOnBlur key is used within the argument of \AA to define an 'on blur' event:

```
\textField[\AA{\AAOnBlur{%
    \JS{app.alert("Thanks for the data, I think!")}}}]
{myText}{1.5in}{12bp}
```

7. **Format:** The format event is the event that occurs when text is entered into a text or combo box; during this event, optionally defined JavaScript code is executed to format the appearance of the text within the field. The \AAFormat key is used within the argument of \AA to define a format event:

```
\textField[\AA{%
    \AAKeystroke{AFNumber_Keystroke(2,0,1,0,"\\u0024",true);}
    \AAFormat{AFNumber_Format(2,0,1,0,"\\u0024",true);}}]
{myText}{1.5in}{12bp}
```

The above example creates a text field which will accept only a number into it and which will format the number into U.S. currency.

8. **Keystroke:** This keystroke event is the event that occurs when individual keystroke is entered into a choice field (list or combo) or a text field; during this event, optionally defined JavaScript can be used to process the keystroke. The `\AAKeystroke` key is used within the argument of `\AA` to define a keystroke event; see the format example above.

9. **Validate:** The validate event is an event for which JavaScript code can be defined to validate the data that has been entered (text and combo fields only). The `\AAValidate` key is used within the argument of `\AA` to define a validate event:

```
\textField[\AA{%
  \AAKeystroke{AFNumber_Keystroke(2,0,1,0,"\\u0024",true);}
  \AAFormat{AFNumber_Format(2,0,1,0,"\\u0024",true);}
  \AAValidate{%
    if (event.value > 1000 || event.value < -1000) {\r\t
      app.alert("Invalid value, rejecting your value!");\r\t
      event.rc = false;\r
    }}
}]{myText}{1.5in}{12bp}
```

10. **Calculate:** The calculate event is an event for which JavaScript code can be defined to make automatic calculations based on entries of one or more fields (text and combo fields only). The `\AACalculate` key is used within the argument of `\AA` to define a calculate event:

```
\textField[\AA{%
  \AAKeystroke{AFNumber_Keystroke(2,0,1,0,"\\u0024",true);}
  \AAFormat{AFNumber_Format(2,0,1,0,"\\u0024",true);}
  \AACalculate{AFSimple_Calculate("SUM",new Array("Prices"));}
}]{myText}{1.5in}{12bp}
```

11. **PageOpen:** (The **PO** key, Table 8.10, PDF 1.5) An action to be performed when the page containing the annotation is opened (for example, when the user navigates to it from the next or previous page or by means of a link annotation or outline item). The action is executed after the page's open action. The `\AAPageOpen` key is used within the argument of `\AA` to define an annotation page open event:

12. **PageClose:** (The **PC** key, Table 8.10, PDF 1.5) An action to be performed when the page containing the annotation is closed (for example, when the user navigates to the next or previous page, or follows a link annotation or outline item). The action is executed before the page's close action. `\AAPageClose` key is used within the argument of `\AA` to define an annotation page close event.

13. **PageVisible:** (The **PV** key, Table 8.10, PDF 1.5) An action to be performed when the page containing the annotation becomes visible in the viewer application's user interface. `\AAPageVisible` key is used within the argument of `\AA` to define an annotation page visible event.

14. **PageInvisible:** (The **PI** key, Table 8.10, PDF 1.5) An action to be performed when the page containing the annotation is no longer visible in the viewer application's user interface. `\AAPageInvisible` key is used within the argument of `\AA` to define an annotation page invisible event.

    Below is a simple example of usage. The actions write to the console.

```
\textField[\AA{%
    \AAPageOpen{console.println("Page \thepage: PO");}
    \AAPageClose{console.println("Page \thepage: PC");}
    \AAPageVisible{console.println("Page \thepage: PV");}
    \AAPageInvisible{console.println("Page \thepage: PI");}
    }]{tf\thepage}{2in}{11bp}
}
```

Additional examples appear in the file `eqforms.tex`.

4.2. Action Types

The following is only a partial listing of the action types, as given in Table 8.36 of the *PDF Reference* [5]. The entire list and the details of usage can be obtained from the *PDF Reference*.

Action Type	Description
GoTo	Go to a destination in the current document
GoToR	Go to a destination in another document
Launch	Launch an application, usually to open a file
URI	Resolve a uniform resource identifier
Named	Execute an action predefined by the viewer
SubmitForm	Send data to a uniform resource locator
JavaScript	Execute a JavaScript script (PDF 1.3)

Examples of each type of action follow.

► GoTo: Go to a (named or explicit) destination within the current document. In this example, we 'go to' the named destination `toc.1`, which references the table of contents pages. This button goes to a *named destination*: Go

```
\pushButton[\CA{Go}\AC{Now!}\RC{to TOC}
    \A{/S/GoTo/D(toc.1)}]{myButton1}{}{10bp}
```

For a named destination, the value of the `/D` key is a string, `(toc.1)` in the above example, that specifies the destination name.

The following is an example of an *explicit destination*: Go

```
\pushButton[\CA{Go}\AC{Now!}\RC{to Page 3}
\A{/S/GoTo/D[{Page3}/Fit]}]{myButton1}{}{10bp}
```

The value of the destination key /D is an array referencing a page number ({Page3}) and a view (/Fit).

For a GoTo action, the first entry in the destination array, {Page3}, is an indirect reference to a page, the notation {Page3} is understood by the distiller. For dvipdfm, use the @page primitive:

```
\makeatletter\def\Page#1{@page#1}\makeatother
\pushButton[\CA{Go}\AC{Now!}\RC{to Page 3}
\A{/S/GoTo/D[\Page3/Fit]}]{myButton1}{}{10bp}
```

pdftex has no mechanism for inserting indirect page references.

See section 8.5.3, 'Go-To Actions', of the *PDF Reference* [5] for details of the syntax of GoTo, and section 8.2.1 for documentation on explicit and named destinations.

► GoToR: Go to a (named or explicit) destination in a remote document. In this example, we 'go to a remote' destination, a *named destination* in another document. Go

```
\pushButton[\CA{Go}\AC{Now!}\RC{to TOC}
    \pushButton[\CA{Go}\AC{Now!}\RC{to TOC}
    \A{/S/GoToR/F(webeqtst.pdf)/D(webtoc)]{myButton2}{}{10bp}
```

This example illustrates an *explicit destination*; the following button jumps to page 3 in another document: Go

```
\pushButton[\CA{Go}\AC{Now!}\RC{to Page 3}
\A{/S/GoToR/F(webeqtst.pdf)/D[2/Fit]}]{myButton2}{}{10bp}
```

The value of the destination key /D is an array referencing a page number and a view (/Fit).

For an *explicit destination*, the *PDF Reference* [5] specifies that the first entry in the destination array should be a page number (as contrasted with an indirect reference to a page number, for the case of GoTo). The destination, /D[2/Fit] would correctly work for distiller, dvipdfm and pdftex.

See section 8.5.3, 'Remote Go-To Actions', of the *PDF Reference* [5] for details of the syntax of GoToR, and section 8.2.1 for documentation on explicit and named destinations.

► Launch: Launch an application ('Open a file'). In this example, we open a TEX file using the application associated with the .tex extension: Go

```
\pushButton[\CA{Go}\AC{Now!}\RC{to TOC}
\A{/S/Launch/F(webeqtst.tex)}]{myButton3}{}{10bp}
```

See section 8.5.3, 'Launch Actions', of the *PDF Reference* [5] for details of the syntax.

► URI: Open a web link. In this example, we go to the Adobe web site: Go

```
\pushButton[\CA{Go}\AC{Adobe!}\RC{To}
\A{/S/URI/URI(http://www.adobe.com/)}]{myButton4}{}{10bp}
```

See section 8.5.3, 'URI Actions', of the *PDF Reference* [5] for details of the syntax.

Acrobat (Adobe Reader) also support open parameters, using these key-value pairs, we can go to a specific page in a PDF on the web, and even search for words, for example Go & Search opens the AeB Manual on the Internet, goes to page 8, and searches for the words AcroTeX, web, and exerquiz.

```
\pushButton[\CA{Go & Search}
\A{/S/URI/URI(http://www.math.uakron.edu/~dpstory/
    acrotex/aeb_man.pdf#page=8&search=AcroTeX web exerquiz)}]
    {myButton4a}{}{10bp}
```

The same can be accomplished using `\setLink`.

► `Named`: Execute a 'named' action (i.e., a menu item). Named actions listed in the *PDF Reference* are `NextPage`, `PrevPage`, `FirstPage` and `LastPage`. A complete list of named actions can be obtained by executing the code `app.listMenuItems()` in the JavaScript console of Acrobat (Pro). Go

```
\pushButton[\CA{Go}\AC{Previous!}\RC{To}
\A{/S/Named/N/PrevPage}]{myButton5}{}{10bp}
```

See section 8.5.3, 'Named Actions,' of the *PDF Reference* [5] for details of the syntax. The named actions listed in the PDF Reference are `NextPage`, `PrevPage`, `FirstPage`, and `LastPage`.

In theory, any menu item can be executed as a named actions; there are several factors to be taken into consideration: (1) Not all menu items available to Acrobat are listed on the menu bar of Adobe Reader, when choosing a name event to use, you should decide if the application executing the named action supports that action; (2) In recent versions, starting with version 7, there have been security restrictions on the execution of menu items, the so-called "white list." Only named actions listed on the white list are allowed to execute. The white list for version 8.0 is

Named Actions on Whitelist

AcroSendMail:SendMail	LastPage	ShowHideToolbarCommenting
ActualSize	NextPage	ShowHideToolbarData
AddFileAttachment	OneColumn	ShowHideToolbarEdit
BookmarkShowLocation	OpenOrganizer	ShowHideToolbarEditing
Close	PageSetup	ShowHideToolbarFile
CropPages	PrevPage	ShowHideToolbarFind
DeletePages	Print	ShowHideToolbarForms
ExtractPages	PropertyToolbar	ShowHideToolbarMeasuring
Find	Quit	ShowHideToolbarNavigation
FindCurrentBookmark	ReplacePages	ShowHideToolbarPageDisplay
FindSearch	RotatePages	ShowHideToolbarPrintProduction
FirstPage	SaveAs	ShowHideToolbarRedaction
FitHeight	Scan	ShowHideToolbarTasks
FitPage	ShowHideAnnotManager	ShowHideToolbarTypewriter
FitVisible	ShowHideArticles	SinglePage
FitWidth	ShowHideBookmarks	Spelling
FullScreen	ShowHideFields	Spelling:Check

Named Actions on Whitelist

GeneralInfo	ShowHideFileAttachment	TwoColumns
GeneralPrefs	ShowHideModelTree	TwoPages
GoBack	ShowHideOptCont	Web2PDF:OpenURL
GoForward	ShowHideSignatures	ZoomTo
GoToPage	ShowHideThumbnails	ZoomViewIn
InsertPages	ShowHideToolbarBasicTools	ZoomViewOut

In addition to the Whitelist for version 8, the following menu items are added for version 9.

Named Actions on Whitelist

Annots:Tool:InkMenuItem	CollectionShowRoot	HandMenuItem
CollectionDetails	DocHelpUserGuide	HelpReader
CollectionHome	GoBackDoc	rolReadPage
CollectionPreview	GoForwardDoc	ZoomDragMenuItem

As mentioned before, some of these are for Acrobat only, others are available for Adobe Reader. I'll let you sort them out. If you try to execute a named action that is not on the white list, the action will silently fail.

► `SubmitForm`: Submit forms Action. In this example, we submit a URL to a CGI, which then sends the requested file back to the browser:

Homepage of D. P. Story Go!

Note: This script no longer works, server-side scripting at the `uakron.edu` server is no longer permitted (for security reasons). The verbatim lisiting of the code as it used to be when it worked.

```
\definePath{\URL}{http://www.math.uakron.edu/~dpstory}
\comboBox[\DV{\URL}\V{\URL}\BG{webyellow}\BC{webgreen}]
{dest}{1.75in}{11bp}{%
    [(\URL)( Homepage of D. P. Story)]
    [(\URL/acrotex.html)( AcroTeX Homepage)]
    [(\URL/webeq.html)( AcroTeX Bundle)]
    [(\URL/acrotex/examples/webeqtst.pdf)(Exerquiz Demo file {(PDF)})]
}\kern1bp\pushButton[\BC{webgreen}\CA{Go!}
\A{/S/SubmitForm/F(http://www.math.uakron.edu/cgi-bin/nph-cgiwrap/%
dpstory/scripts/nph-redir.cgi)/Fields[(dest)]/Flags 4}]
{redirect}{33bp}{11bp}
```

See section 8.6.4 of the *PDF Reference* [5] for details of the syntax for 'Submit Actions'.

► `JavaScript`: Execute a JavaScript action. This is perhaps the most important type of action. In this example, the previous example is duplicated using the `Doc.getURL()` method, we don't need to submit to a CGI.

Homepage of D. P. Story Go!

```
\definePath{\URL}{http://www.math.uakron.edu/~dpstory}
\comboBox[\DV{\URL}\V{\URL}\BG{webyellow}\BC{webgreen}]
```

```
{dest}{1.75in}{11bp}{%
    [(\URL)( Homepage of D. P. Story)]
    [(\URL/acrotex.html)( AcroTeX Homepage)]
    [(\URL/webeq.html)( AcroTeX Bundle)]
    [(\URL/acrotex/examples/webeqtst.pdf)(Exerquiz Demo file {(PDF)})]
}\kern1bp\pushButton[\BC{webgreen}\CA{Go!}
\A{\JS{%
    var f = this.getField("dest");\r
    app.launchURL(f.value,false);
}}]{redirect}{33bp}{11bp}
```

Note the use of the convenience command \JS, defined in the insldjs package, it expands to the correct syntax: /S/JavaScript/JS(#1), where #1 is the argument of \JS.

Most all actions can be performed using JavaScript, the reader is referred to the *JavaScript for Acrobat API Reference* [4].

5. JavaScript

Acrobat JavaScript is the cross-platform scripting language of the Acrobat suite of products. For Acrobat 5.0 or later, Acrobat JavaScript based on JavaScript version 1.5 of ISO-16262 (formerly known as ECMAScript), and adds extensions to the core language to manipulate Acrobat forms, pages, documents, and even the viewer application.

Web-based references to core JavaScript are the *Core JavaScript Guide* [1] and the *Core JavaScript Reference* [2]. For Acrobat JavaScript, we refer you to the *Developing Acrobat Applications using JavaScript* [3] and the *JavaScript for Acrobat API Reference* [4].

5.1. Support of JavaScript

The AcroTEX eDucation Bundle has extensive support for JavaScript, not only for JavaScript executed in response to a field trigger, but for document level and open page actions as well. As the topic of this document is eForm support, the reader is referred to the documentation in the insdljs package, which is distributed with the AcroTEX Bundle.

• The Convenience Command \JS

The syntax for writing JavaScript actions is

```
\pushButton[\A{/S/JavaScript/JS(⟨script⟩)}]
    {jsEx}{22bp}{11bp}
```

Notice the code is enclosed in matching parentheses. As noted earlier, AcroTEX defines the command \JS as a convenience for this very common actions; the above example becomes:

```
\pushButton[\A{\JS{⟨script⟩}}]{jsEx}{22bp}{11bp}
```

The code is now enclosed in matching braces.

• Inserting Simple JavaScript

Actions are introduced into a field command through its optional first parameter. JavaScript actions, in particular, can be inserted by a mouse up[3] action, for example, using the \A and \JS commands.

The "environment" for entering JavaScript is not a verbatim environment: '\' is the usual TEX escape character and expandable commands are expanded; active characters are expanded (which is usually not what you want); and primitive commands appear verbatim (so you can use, for example, '{' and '}). Within the optional argument, the macro \makeJSspecials, which can be redefined, is expanded; the macro makes several special definitions: (1) it defines \\ to be '\\'; (2) defines \r to be the JavaScript escape sequence for new line; and (3) defines \t to be the JavaScript escape sequence for tab.

Example 12. Sum

The verbatim listing for this button is

```
\pushButton[\CA{Sum}\A{\JS{%
    var n = app.response("Enter a positive integer",
        "Summing the first \\"n\\" integers");\r
    if ( n != null ) {\r\t
        var sum = 0;\r\t
        for ( var i=1; i <= n; i++ ) {\r\t\t
            sum += i;\r\t
        }\r
    app.alert("The sum of the first n = " + n
        + " integers is " + sum + ".", 3);
    }
}}]{jsSum}{22bp}{11bp}
```

Code Comments. Within the JavaScript string, we want literal double quotes ", to avoid " being interpreted as the end of the string (or the beginning of a string) we have to double escape the double quotes, as in \\". (This is not necessary when entering code in the JavaScript editor if you have the Acrobat application.) I try to write JavaScript that I can easily read, edit, and debug in the JavaScript editor (available in the full Acrobat application); for this reason, I've added in new lines and tabbing (\r and \t). Many people, however, have only the Adobe Reader and cannot see their code to debug it; in this case, the formatting is really not needed.

Needless to say, the following sample will not compile because we do not have matching braces.

```
\pushButton[\A{\JS{var x = "{"}}]{jsBrace}{22bp}{11bp}
```

The work around here is

```
\pushButton[\A{\JS{var x = "\jslit\{"}}]{jsBrace}{22bp}{11bp}
```

[3] Other types of possible actions are discussed and illustrated in 'Actions' on page 18.

In the above work about, the \jslit command (for JavaScript literal) is used. This command is defined only within the optional arguments of a form field. The definition of \jslit is \let\jslit\string

• Inserting Complex or Lengthy JavaScript

For JavaScript that is more complex or lengthy, the insdljs Package, distributed with the AcroTEX Bundle, has the verbatim defineJS environment. Details and idiosyncracies of this environment are documented in 'The defineJS Environment' on page 51. The example given in Example 6 will suffice; the verbatim listing is reproduced here for convenience.

▶ First, we define the JavaScript action and name it \getComboJS for the button (prior to defining the field, possibly in the preamble, or in style files):

```
\begin{defineJS}{\getComboJS}
    var f = this.getField("myCombo");
    var a = f.currentValueIndices;
    if ( a == -1 )
        app.alert("You've typed in \\"" + f.value +"\\".");
    else
        app.alert("Selection: " + f.getItemAt(a, false)
            + " (export value: " + f.getItemAt(a, true)+").");
\end{defineJS}
```

There is no need for the \r and \t commands to format the JavaScript; the environment obeys lines and spaces; contrast this example with **Example 12**, page 26.

Now we can define our fields, a combo box (not shown) and button, in this example. It is the button that uses the JavaScript defined above.

```
\pushButton[\BC{0 .6 0}\CA{Get}\AC{Combo}\RC{Box}
    \A{\JS{\getComboJS}}]{myComboButton}{33bp}{11bp}
```

Within the argument of \JS we insert the macro command, \JS{\getComboJS} for our JavaScript defined earlier in the defineJS environment

The demo file aeb_links.pdf, with source attached, is found on the AcroTEX Blog website.

5.2. Defining JavaScript Strings with \defineJSStr

The command \defineJSStr is used to define JavaScript strings, such as in dialog boxes. The syntax for this command is

\defineJSStr{\⟨*CMD*⟩}{⟨*JS_string*⟩}

Parameter Description: The parameter \⟨*CMD*⟩ is a command to be define by `\defineJSStr`, for example, `\myMessage`, and ⟨*JS_string*⟩ is the JavaScript string to be defined as the expansion of the \⟨*CMD*⟩.

Command Description: `\defineJSStr` executes `\xdef#1{"#2"}`, so the JavaScript is expanded at the time of definition. (Note the enclosing double quotes) Before the expansion occurs, however, there are a number of definitions that occur locally:

- `\uXXXX` is recognized as a unicode escape sequence. So, within the JavaScript string, unicode can be entered directly, for example, `\u00FC` is the u-umlaut.
- Backslash is still the tex escape character, so any commands in the JavaScript string get expanded. You can delay the expansion by using `\protect`. Expansion occurs when the tex compiler actually expands \⟨*CMD*⟩.
- `\r` (carriage return), `\n` (line feed) and, `\t` (tab) can be used to format the message, as desired.
- Use the `\cs` command to write a word containing a literal backslash in it; for example, to get `\LaTeX` to appear in a JavaScript string, you must type `\cs{LaTeX}` in the JavaScript string.
- The JavaScript string is enclosed in double quotes ("), if you want a literal double quote, use \\" to get a literal double quote to appear in a JavaScript string. For example,

```
\defineJSStr{\myMessage}
    {My name is \\"Stan\\" and I'm \\"the man.\\"}
```

- The command `\jslit` is recognized within the JavaScript string. Using `\jslit` (short for JavaScript literal), you can insert, for example, unbalanced braces:

```
\defineJSStr{\myMessage}
    {You forgot the left brace \\"\jslit\{\\",
     please insert it.}
```

  The definition of `\jslit` is `\let\jslit\string`.

The design decision to automatically insert the double quotes in the definition of the string has its faults. When you want to break the string to insert dynamic content, you must always be aware of the definition `\xdef#1{"#2"}` contained with the definition of `\defineJSStr`. To illustrate, we return to an earlier example presented in Inserting Complex or Lengthy JavaScript, on page 27.

We begin by defining some JavaScript strings. Note that in each of these two definitions, the leading and trailing double quote (") is missing (these are the ones inserted automatically). The definitions look a bit strange because we break the string to insert dynamic content (`f.value`, `f.getItemAt(a, false)`, etc.), then continue on with the string from there.

```
\defineJSStr\myAlerti{You've typed in \\"" + f.value +"\\".}
\defineJSStr\myAlertii{Selection: " + f.getItemAt(a, false)
   + " (export value: " + f.getItemAt(a, true)+").}
```

We use the `defineJS` environment, after setting the escape code to `@`. The `defineJS` is a fully verbatim environment, the escape character cannot be changed to \, but it may be changed to another character, such as `@`.

```
\begin{defineJS}[\catcode`\@=0\relax]{\getComboJS}
var f = this.getField("myCombo");
var a = f.currentValueIndices;
if ( a == -1 )
    app.alert(@myAlerti);
else
    app.alert(@myAlertii);
\end{defineJS}
```

Then again, the use of `\defineJSStr` is not required, it is a convenience for creating JavaScript strings, especially ones with embedded Unicode. The previous example could have been done by defining `\myAlerti` and `\myAlertii` by

```
\newcommand\myAlerti{"You've typed in \"" + f.value +"\"."}
\newcommand\myAlertii{"Selection: " + f.getItemAt(a, false)
   + " (export value: " + f.getItemAt(a, true)+")."}
```

Note the *presence* of the leading and trailing double quotes.

6. The useui option: A User-Friendly Interface

To use the "user-friendly" interface, the `useui` option must be taken. The key-value pairs described below are enclosed as the argument of the special `\ui` key. For example,

```
\pushButton[\ui{%
    bordercolor={1 0 0},bgcolor={0 1 0},
    textcolor={1 0 0},align={right},
    uptxt={Push Me},
    js={app.alert("AcroTeX rocks!")}
}]{pb1}{}{11bp}
```

You can develop your own set of appearances and use the `presets` key to conveniently set these. For example,

```
\def\myFavFive{%
    bordercolor={1 0 0},bgcolor={0 1 0},
    textcolor={1 0 0},align={right},
    uptxt={Push Me}
}
```

Later, a push button can use this preset, like so,

```
\pushButton[\ui{presets=\myFavFive,
    js={app.alert("AcroTeX rocks!")}}]{pb1}{}{11bp}
```

which produces Push Me [4]

You can mix your \myFavFive with different key-value pairs, such as a JavaScript action.

6.1. The Appearance Tab

We set these key-value pairs to model the user interface of Acrobat.

The key is border. In the case of a link, this is the Link Type: Visible Rectangle or Invisible Rectangle. For forms, this key has no counterpart in the user interface. If you set border equal to invisible, that will set border line width to zero \W{0}.

`border=visible|invisible`

Command Description: Used with link annotations and determines whether the border surrounding the bounding box of the link is visible. If this key is not specified, the eforms follows the rule: If colorlinks option of hyperref is used, the border is invisible; otherwise, it is visible (and the default linewidth is 1). Use the border key to override this behavior.

`linewidth=thin|medium|thick`

Command Description: The linewidth of the border around a link or a form. The user interface choices are thin, medium, and thick. This key-value is ignored if the document author has set the border to invisible.

`highlight=none|invert|outline|inset|push`

Command Description: The highlight type for links and forms, choices are none, invert, outline, inset and push. The term inset is used with links, and push is used with forms. They each have the same key value pair.

`bordercolor=`⟨*num*⟩␣⟨*num*⟩␣⟨*num*⟩

Command Description: The color of the border, when visible, in RGB color space. For example, bordercolor=1 0 0, is the color red.

`linestyle=solid|dashed|underlined|beveled|inset`

Command Description: The line style of the border, possible values are solid,dashed, underlined, beveled,and inset. Links do not support the beveled option.

`dasharray=`⟨*num*⟩[␣⟨*num*⟩]

Command Description: When a line style of dashed is chosen, you can specify a dash array. The default is 3.0, which means a repeating pattern of 3 points of line, followed by 3 points of space. A value of dasharray=3 2 means three points of line, followed

[4] The reader is reminded once again that the author has no understanding of colors.

by two points of space. When this key is used without a value, the value is 3.0. When the dashed key is not present, 3.0 is used.

`linktxtcolor=⟨named_color⟩`

Command Description: Set the color of the link text. Ignored if the colorlinks option of hyperref has not been taken. The value of linktxtcolor is a named color. For example, linktxtcolor=red. The default is \@linkcolor from hyperref. This default can be changed by redefining \@linkcolor, or redefining \defaultlinkcolor. If linktxtcolor={} (an empty argument), or simply linktxtcolor, no color is applied to the text, the color of the text will be whatever the current color is.

`annotflags=hidden|print|-print|noview|lock`

Command Description: This is a bit field, possible values are hidden, print, -print, noview, and lock. *Multiple values can be specified.* The values are "or-ed" together. Most all forms are printable by default. If you don't want a form field to print specify -print. For example, annotflags={-print,lock} makes the field not printable and is locked, so the field cannot be moved through the UI.

```
fieldflags=readonly|required|noexport|multiline|password|
     notoggleoff|radio|pushbutton|combo|edit|
     sort|fileselect|multiselect|nospellcheck|
     noscrolling|comb|radiosinunison|commitonchange|
     richtext
```

Command Description: There are a large number of field flags (Ff) that set a number of properties of a field. This is a multiple-selection key as well. The values are "or-ed" together.

Normally, a document author would not specify radio, pushbutton or combo. These properties are used by eforms to construct a radio button field, a push button and a combo box. The others can be used as appropriate.

`maxlength=⟨num⟩`

Command Description: Use maxlength to limit the number of characters input into a text field. Example: maxlength=12. When the fieldflags is set to comb, the value of maxlength determines the number of combs in the field.

`tooltip=⟨string⟩`

Command Description: Enter a text value to appear as a tool tip. A tool tip is text that appears in a frame when the user hovers the mouse over the field. The link annotation does not have a tool tip feature. Enclose in parentheses if the text string contains a comma; for example, tooltip={Hi, press me and see what happens!}. The tooltip key obeys the unicode option. If the unicode option of hyperref is in effect, then setting

```
tooltip = {J\"{u}rgen, press me and see what happens!}
```

yields a tool tip of “Jürgen, press me and see what happens!”

`default=⟨string⟩`
`value=⟨string⟩`

Command Description: Set default value of a field (text, list, combobox) using the `default` key. The default value is the value used for the field when the field is reset. Example: `default=Name`.

The `value` key is used to set the current value of a field (text, list, combobox). Example: `value=AcroTeX`.

These two keys obey the `unicode` option. If the unicode option of hyperref is in effect, then setting `value = \texteuro\ 1 000 000` sets the (initial) value of this field to “€ 1 000 000”.

`rotate=0|90|180|270`

Command Description: Set the orientation of the field, values are 0, 90, 180 and 270. If 90 or 270 are chosen, the height and width of the field need to be reversed. This is not done automatically by eforms

`bgcolor=⟨num⟩␣⟨num⟩␣⟨num⟩`

Command Description: The background color of a form field. This is a RGB color value.

`uptxt=⟨string⟩`
`downtxt=⟨string⟩`
`rollovertxt=⟨string⟩`

Command Description: The normal (mouse up), mouse down and rollover text for a button field. All three of these keys obey the `unicode` option. If the `unicode` option of hyperref is in effect, then setting `uptxt = J\"{u}rgen` yields a normal caption of “Jürgen” on the button.

Push buttons only. The following list of keys are used for creating custom appearances on button faces. Acrobat Distiller required for this set. The example files `eqforms.pdf` and `eqforms_pro.pdf` illustrate the creation of icons as button appearances. In the latter PDF, `eqforms_pro.pdf`, Acrobat Distiller is required to be the PDF creator.

`normappr=⟨string⟩`
`rollappr=⟨string⟩`
`downappr=⟨string⟩`

Command Description: The normal, rollover, and down appearances of the button face icon. The value of each key is an indirect reference to a form XObject. Normally, you can use the `graphicxsp` package to embed graphics and give a symbolic name which is used as the value of these keys.

```
layout=labelonly|icononly|icontop|iconbottom|
       iconleft|iconright|labelover
```

Command Description: The value of this key determines the layout of the icon relative to the label (or caption). The default is `labelonly`, if if you define icons, you need to set layout to something other than `labelonly`.

```
scalewhen=always|never|iconbig|iconsmall
```

Command Description: The value of this key tells when to scale the icon. The `iconbig` scales the icon when it is too big for the bounding rectangle; while `iconsmsll` scales the icon when it is too small for the bounding rectangle. The default is `always`.

```
scale=proportional|nonproportional
```

Command Description: This parameter sets the scale type, either `proportional` scaling, where the aspect ratio of the icon is preserved; or `nonproportional` scaling is used. The default is `proportional`.

position=⟨*x*⟩␣⟨*y*⟩

Command Description: Both ⟨*x*⟩ and ⟨*y*⟩ are numbers between 0 and 1, inclusive, and separated by a space (not a comma). They indicate the fraction of the left over space to allocate at the left and bottom of the icon. A value of `{0.0 0.0}` positions the icon at the bottom-left corner; a value of `{0.5 0.5}` centers it within the rectangle. This entry is only used of the icon is scaled proportionally. The default is `{0.5 0.5}`.

```
fitbounds=true|false
```

Command Description: A Boolean value, if `true`, indicates that the button appearance should be scaled to fit fully within the bounds of the field's bounding rectangle without taking into consideration the line width of the border. The default is `false`. `fitbounds` is the same as `fitbounds=true`.

Check boxes and Radio Buttons Only. The following list of keys are used for creating custom appearances on check boxes and radio buttons. Acrobat Distiller required for this set. The example files `eqforms.pdf` and `eqforms_pro.pdf` illustrate the creation of these appearances.. In the latter PDF, `eqforms_pro.pdf`, Acrobat Distiller is required to be the PDF creator.

```
appr={ norm={on={⟨normOnAppr⟩},off={⟨normOffAppr⟩}},
       down={on={⟨downOnAppr⟩},off={⟨downOffAppr⟩}},
       roll={on={⟨rollOnAppr⟩},off={⟨rollOffAppr⟩}}}
```

Command Description: The norm key is the normal appearance of the button; it has two appearances, the on and the off appearances. The on and off are indirect references to a form XObject. The other two keys, down and roll, are the down and rollover appearances, respectively; they have the same structure as norm does.

If appr is not specified, then, by default, the usual appearances of the buttons are used, as provided by Acrobat/AR.

The down and roll are optional, if you use appr at all, you should specify the norm appearance, both on and off appearances.

```
align=left|centered|right
```

Command Description: The type of alignment of a text field. Permitted values are left, centered, and right.

```
textfont=⟨font_name⟩
textsize=⟨num⟩
textcolor=⟨num⟩[␣⟨num⟩␣⟨num⟩[␣⟨num⟩]]
```

Command Description: The key textfont is the text font to be used with the text of the field, while textsize is the text size to be used. A value of 0 means auto size. The color of the text in the field. This can be in G, RGB or CMYK color space by specifying 1, 3 or 4 numbers between 0 and 1.

```
autocenter=yes|no
```

Command Description: This is a feature of eforms. Use autocenter=yes (the default) to moderately center the bounding box, and use autocenter=no otherwise.

```
presets=\CMD
```

Command Description: Set presets from inside a \ui argument. The value of \ui must be a user defined command, which expands to a comma-delimited list of ui-key-value pairs.

Example 13. Use the presets key to place pre-defined key-value pairs into the option argument of a link. Define a command,

```
\def\myUIOpts{%
    border=visible,linktxtcolor=blue,
    linewidth=medium,highlight=outline,
```

```
        linestyle=dashed,bordercolor={1 0 0},
        js={app.alert("AcroTeX rocks!")}
    }
```

Later, we can type,

```
\setLink[\ui{presets={\myUIOpts}}]{Press Me Again!!}
```

```
symbolchoice=check|circle|cross|diamond|square|star
```

Command Description: Used with a checkbox or radio button field. This sets the symbol that appears in the field when the box is checked. Choices are `check`, `circle`, `cross`, `diamond`, `square`, and `star`.

6.2. The Action Tab

There are several common actions that are supported through the user-friendly interface, these are `goto` actions, and JavaScript actions.

```
goto={KV-pairs}
```

Command Description: This key incorporates jumps to pages and destinations within the current PDF file, and to pages and destinations to another PDF file. these are

Key-Value Pairs: There are a number of key-value pairs that are recognized, `file`, `targetdest`, `labeldest`, `page`, `view`, and `open`. A brief description of each follows.

1. `file`: Specify a relative path to the PDF file. This will work on the Web if the position is the same relative to the calling file. If the `file` key is not present, the jump is to a page or destination in the current file.
2. `url`: This key is used to create a weblink, similar to what `\href` does. The value of this key is a url (`http`, `https`, `mailto`, etc.). If the `url` key is present, only the `openparams` key is recognized.
3. `openparams`: Open parameters that should be included with the URL, as passed by the `url` key. These parameters are key value pairs `key=value` and are separated by an ampersand (&). See *Parameters for Opening PDF Files* for more information, examples are found below.
4. `targetdest`: Jump to a target, perhaps created by `\hypertarget`. For example, if we say `\hypertarget{acrotex}{Welcome!}`, we jump to the `acrotex` named destination by specifying `targetdest=acrotex`.
5. `labeldest`: Same as `targetdest`, but we jump to a destination specified by a latex label. For example, if we type `\section{AcroTeX}\label{acrotex}`, we can jump to this section by specifying `labeldest=acrotex`.

6. page: The page number to which the goto action is to jump. If we set page=1, we will jump to the first page of the document.
7. view: The view can be set when the page key is used. Possible values are fitpage, actualsize, fitwidth, fitvisible, and inheritzoom. These terms correspond to Acrobat's UI. When jumping to a destination, the view is set by the destination code.
8. open: This key is used when you specify the file key. The open key determines if a new window is opened or not when the PDF viewer jumps to the file. Possible values are userpref (use user preferences), new (open new window), existing (use the existing window).

Example 14. The following are examples of the goto key.

- AeB Manual

```
\setLink[\ui{goto={file=aeb_man.pdf,page=8,%
    view=fitwidth}}]{AeB Manual}
```

  This link should work on your local hard drive and it should work on the web, from within a web browser, assuming aeb_man.pdf is in the same folder as eformman.pdf.

- AeB Manual on Web

```
\setLink[\ui{%
    goto={url=http://www.math.uakron.edu/~dpstory/%
        acrotex/aeb_man.pdf,%
    openparams={page=8&search=AcroTeX web exerquiz}}}
        ]{AeB Manual on Web}
```

  Here, we open the AeB Manual that is on the web, go to page 8, and search for the words AcroTeX, web, and exerquiz. Notice that we don't have to do anything special with the tilde (~) or the sharp (#), both of these are handled by the eforms package.

`js={⟨script⟩}`

Command Description: A general purpose key to execute JavaScript actions on a mouse up trigger. The argument is a JavaScript text string, for example,

```
js={app.alert("Hello World!")}
```

The value of js may be a macro containing JavaScript, which would include a macro created by the defineJS environment of insdljs.

```
mouseup={⟨script⟩}
mousedown={⟨script⟩}
onenter={⟨script⟩}
onexit={⟨script⟩}
onfocus={⟨script⟩}
onblur={⟨script⟩}
format={⟨script⟩}
keystroke={⟨script⟩}
validate={⟨script⟩}
calculate={⟨script⟩}
pageopen={⟨script⟩}
pageclose={⟨script⟩}
pagevisible={⟨script⟩}
pageinvisible={⟨script⟩}
```

Command Description: These are all additional actions (AA) of a form field, which take as their values JavaScript code (⟨*script*⟩).

- `mouseup`: Executes its code with a mouse up event. If there is a JavaScript action defined by the `js` key (or the \A key), the `js` (\A) action is executed.
- `mousedown`: Executes its when the mouse is hovering over the field and the user clicks on the mouse.
- `onenter`: Executes its code when the user moves the mouse into the form field (the bounding rectangle).
- `onexit`: Executes its code when the user moves the mouse out of the form field (the bounding rectangle).
- `onfocus`: Executes its code when the user brings the field into focus.
- `onblur`: Executes its code when the user brings the field loses focus (the user tabs away from the field, or click outside the field).
- `format`: JavaScript to format the text that appears to the user in a text field or editable combo box.
- `keystroke`: JavaScript to process each keystroke in a text field or editable combo box.
- `validate`: JavaScript to validate the committed data input into a text field or editable combo box.
- `calculate`: JavaScript to make calculations based on the values of other fields.
- `pageopen`: JavaScript that executes when the page containing the field is opened.
- `pageclose`: JavaScript that executes when the page containing the field is closed.

- pagevisible: JavaScript that executes when the page containing the field first becomes visible to the user.
- pageinvisible: JavaScript that executes when the page containing the field is no longer visible to the user.

6.3. The Signed Tab

A signature field has a Signed tab. On that tab is an option to mark a set of fields as readonly (locked). The locked key controls that option.

lock={⟨*PDF KV-pairs*⟩}

Command Description: The lock key is used with signature fields, currently, there is no nice user interface to this key. Typical entries are

```
lock={/Action/All}          % lock all fields in the doc
lock={/Action/Include       % lock all fields listed in Fields
    /Fields [(field1)(field2)...]}
lock={/Action/Exclude       % lock all fields not listed in Fields
    /Fields [(field1)(field2)...]}
```

Another option that is included in the Signed tab is titled “This script executes when field is signed.”

This is an option that, through the user interface, is mutually exclusive from locking fields. This option is implemented through the format event; thus, to populate this option with JavaScript use the format key. For example,

```
format={app.alert("Thank you for signing this field.");}
```

Setting the Tab Order

The taborderPkg package is an internal AeB package that is called by both the eforms and the annot_pro packages. The taborder package sets the tab order for form fields and link annotations (when the link is created by the command \setLink, defined in the eforms package. The package works for all drivers when setting tab order by column, row, or widget order. For setting tabbing order by structure, only documents generated using the **pdfmark** are supported; those using the dvips or dvipsone driver along with Adobe Distiller.

7. Setting the Tab Order

The tabbing order of the fields is usually the order in which the fields were created. In rare cases, it may be desirable to set the order to one of the orders defined by the PDF Reference.

```
\setTabOrder{c|C|r|R|s|S|w|W|a|A|unspecified}
```

Command Description: Command Description: This command is page oriented, it sets to the tab order of fields on the page the TEX compiler executes this command. The permissible values of the parameter are described below, taken verbatim from the *PDF Reference*, the cross-references that appear in the descriptions are references to the *PDF Reference* document.

- c|C (column order): "Annotations are visited in columns running vertically up and down the page. Columns are ordered by the Direction entry in the viewer preferences dictionary (see Section 8.1, 'Viewer Preferences'). The first annotation visited is the one at the top of the first column. When the end of a column is encountered, the first annotation in the next column is visited."
- r|R (row order): "Annotations are visited in rows running horizontally across the page. The direction within a row is determined by the Direction entry in the viewer preferences dictionary (see Section 8.1, 'Viewer Preferences'). The first annotation visited is the first annotation in the topmost row. When the end of a row is encountered, the first annotation in the next row is visited."
- s|S (structure order): "Annotations are visited in the order in which they appear in the structure tree (see Section 10.6, "Logical Structure"). The order for annotations that are not included in the structure tree is application-dependent."
- w|W (version 9.0, widget order): "Widget annotations are visited in the order in which they appear in the page Annots array, followed by other annotation types in row order."
- a|A (version 9.0, annotations array order): "All annotations are visited in the order in which they appear in the page Annots array." (In version 9.0, this key is not implemented.)

- `unspecified|empty` The tab order follows the order of the annotations as listed in the Annots array. For LATEX, this is the order in which the annotations were created. You get the same result if the argument is left empty `\setTabOrder{}`, or if `\setTabOrder` is not used at all. If an unrecognized argument is passed to `\setTabOrder`, `unspecified` is used.

The behavior of tabbing has changed over the years; documentation of tabbing behavior is given in the *Adobe Supplement to the ISO 32000, BaseVersion 1.7, ExtensionLevel 3*.[5] See the section Errors and Implementation Notes. Annotations include things like form fields (widget annotations), links (link annotations) and the various types of comment annotations. See section 8.4.5 of the PDF Reference.

The `\setTabOrder` command is available for users of pdftex and dvipdfm, as well as users of dvipsone and dvips (with distiller); for row, column, and widget (version 9 or later), the PDF viewer does all the work on tabbing, for tabbing using structure, one necessarily needs structure, otherwise, the tabbing follows row order. For users of Adobe Distiller, the `taborder` package provides two ways for defining the structure order; on any page in which structure order is used, use only one of the following commands:

```
\setTabOrderByList
\setTabOrderByNumber
```

7.1. Using `\setTabOrderByList`

We illustrate with a simple example, followed by a verbatim listing of the code, and a discussion afterward. We begin by placing two text fields in a row; normally,we would tab from the first one created by the TEX compiler to the next one created. We use structure to reverse the order of tabbing.

text1

text2

The verbatim listing of the above form fields follows:

```
\setTabOrder{s}     % set tab order to structure
\setTabOrderByList  % the default initially

\textField[\V{text1}\objdef{otext1}]{text1}{1.25in}{11bp}\\[3bp]
\textField[\V{text2}\objdef{otext2}]{text2}{1.25in}{11bp}

\setStructTabOrder{% The list of the fields in the desired order
    {otext2}
    {otext1}
}
```

We begin by specifying `\setTabOrder{s}` structure tab order. In the optional argument of the two text fields, we specify an object name for each. These names must be unique

[5] http://www.adobe.com/devnet/acrobat/pdfs/PDF3200_2008.pdf

throughout the whole document; they are used to reference the fields when setting up the tabbing order.

The `\setStructTabOrder` is used to set up the tabbing order, its arguments (enclosed in braces) consists of a list of object names (which must exist on the current page). The order of the object names is the order of visitation when you tab. PDF objects not referenced are visited last after the structure tabbing is complete.

After all annotations have been created on a page, we use the `\setStructTabOrder` to actually set the tab order; this is none by simply listing the object names, in the desired order, of the annotations you want included in the tabbing order. These annotations can be fields, links, and markup comments, like sticky notes.

The syntax for `\setStructTabOrder` is

```
\setStructTabOrder{%
    [type=⟨type⟩,title=⟨title⟩]{⟨oRef_1⟩}
    [type=⟨type⟩,title=⟨title⟩]{⟨oRef_1⟩}
    ...
    [type=⟨type⟩,title=⟨title⟩]{⟨oRef_n⟩}
}
```

Each argument has an optional argument, the required argument (⟨*<oRef_i>*⟩) is an object name of a previously defined PDF object, such as a form field (widget), a link, or an annotation. The optional argument takes two optional key-value pairs: (1) The type is a declaration of the type the PDF object is, the default is `Form` (you can use `Link` if its a link, and `Annot` if its a comment); (2) `title` is the title of the structure, the value of title appears in the Tags panel of the Acrobat user interface. The default title is to have no title.

The demo file is `settaborder.pdf` for these tabbing features, including tabbing using structure, has its source file attached to the PDF file. The file is posted one the AeB Blog.

7.2. Using `\setTabOrderByNumber`

An alternate method for setting tab order by structure is to directly enter the tab order into the optional argument of the field, link, or comment annotation.

text3

text3

The verbatim listing of the above form fields follows:

```
\setTabOrder{s} % set tab order to structure
\setTabOrderByNumber

\textField[\V{text3}\objdef{otext3}\taborder{1}]
    {text3}{1.25in}{11bp}\\[3bp]
\textField[\V{text3}\objdef{otext4}\taborder{0}]
    {text4}{1.25in}{11bp}
```

Note the user of the `\objdef` and `\taborder` keys. The latter is used to set the order of tabbing.

Important: When setting tab order, there must be an object with `\taborder{0}`; from what I've been able to observe, if no PDF object has tab order zero, the tabbing reverts to what is listed in the Annots array, which is the order the PDF objects were created. If you specify 0, 0, 1, 2, 3…, then the two PDF objects with tab order of 0 are visited in the order they were created. Similarly, for the other tab values. A tab order of 0, 2, 3, 4…seems to work as well. The object labeled 2 will be visited after the object labeled 0.

The demo file is `settaborder1.pdf` for these tabbing features, including tabbing using structure, has its source file attached to the PDF file. The file is posted one the AeB Blog.

Document and Page JavaScript

The insdljsPkg package provides support to LaTeX in four areas:

1. for embedding document level JavaScript into the PDF file created from a LaTeX source, the `insDLJS` environment.
2. for creating open page actions that are executed when the document is first opened to the first page, the `\OpenAction` command.
3. for writing JavaScript code in an environment that preserves the formatting of the code, this is the `defineJS` environment.
4. for executing JavaScript code once to perform post distill tasks, this is the `execJS` environment. This environment works only for document authors that use Acrobat/Acrobat Distiller to create PDF files.

This package defines a new environment, `insDLJS`, used for inserting Acrobat JavaScript into a PDF file created from a LaTeX source. This package works correctly for users of pdftex (and luatex), dvipdfm, dvipdfmx, and `xetex`. For those that use the Acrobat Distiller (specifically, those that use either dvips or dvipsone to produce a postscript file, which is then distilled), you are required to have Acrobat 5.0 (or later).

8. Package Options

The insdljs supports five common "drivers": `dvipsone`, `dvips`, `pdftex` (including the executable lu(la)tex), `dvipdfm`, `dvipdfmx`, `xetex`, and `textures`. When using `dvipsone` or `dvips`, Acrobat Distiller and Acrobat (version 5.0 or later) are required to embed the JavaScripts at the document level. The other drivers have primitives that allow the embedding of the JavaScripts.

Other options are discussed in the following paragraphs.

`nodljs` turns off the embedding of the document level JavaScript. This might be useful, for creating a paper document that is not interactive. For a non-interactive paper document, no JS is needed.

`execJS` is a very useful option/feature if you know how to use it. Any JavaScript that is written in an `execJS` environment is executed once when the document is first opened in Acrobat, then discarded. AeB uses this for post-distillation document processing. The default is that the JavaScript in an `execJS` environment is not executed; using this option turns on this feature.

9. The `insDLJS` and `insDLJS*` Environments

These are the main environments defined by this package. There are two forms of the document level environment, the `insDLJS` and the `insDLJS*`. First, we discuss what a document JavaScript is.

9.1. What is Document Level JavaScript?

The document level is a location in the PDF document where scripts can be stored. When the PDF document is opened, the document level functions are scanned, and any "exposed script" is executed.

Normally, the type of scripts you would place at the document level are general purpose JavaScript functions, functions that are called repeatedly or large special purpose functions. Functions at the document level are known throughout the document, so they can be called by links, form buttons, page open actions, etc.

Variables declared within a JavaScript function have local scope, they are not known outside that function. However, if you can declare variables and initialize them at the document level outside of a function, these variables will have document wide scope. Throughout the document, the values of these global variables are known. For example, suppose the following code is at the document level:

```
var myVar = 17;       // defined outside a function, global scope
function HelloWorld()
{
    var x = 3;        // defined inside a function, local scope
    app.alert("AcroTeX, by Hech!", 3);
}
```

Both the function `HelloWorld()` and the variable `myVar` are known throughout the document. The function `HelloWorld()` can be called by a mouse up button action; some form field, executing some JavaScript, may access the value of `myVar` and/or change its value. The variable `x` is not known outside of the `HelloWorld()` function.

9.2. The insDLJS Environment

The `insDLJS` is the simplest of the two environments. Any material within the environment, eventually ends up in the DLJS section of the PDF document.

The environment takes the *base_name* and writes the file ⟨*base_name*⟩`.djs`. This file contains a verbatim listing of the JavaScript within the environment, plus some lines that change catcodes. The file is then input into the document at `\AtBeginDocument`.

The case of `dvipsone` and `dvips` is a little different. A ⟨*base_name*⟩`.djs` is written and input back, and a second file ⟨*base_name*⟩`.fdf` is written. The second file is later input into the PDF document after distillation.

The syntax of usage for this environment, which takes three arguments, is given next.

```
\begin{insDLJS}[js_var]{base_name}{script_name}
...
    ⟨JavaScript functions or exposed code⟩
...
\end{insDLJS}
```

Environment Description: JavaScript code is written within the `insDLJS` environment. The code is stored as document-level JavaScript, and is global to the document. Func-

tions and variables defined at the top-most level are known to other form elements in the document.

The insDLJS is a verbatim environment, with backslash (\) and percentage (%) maintaining their usual LaTeX meaning. Commands defined in the LaTeX source file, therefore, are expanded before the JavaScript is embedded in the PDF file. The left and right braces are set to normal characters, so the commands can't have any argument, they should be just text macros.

Parameter Description: The environment takes three parameters, the first is optional, but required when using the Acrobat Distiller.

[*js_var*] is an optional parameter *was required* for the dvipsone and dvips options; otherwise it is ignored. Its value must be the name of one of the functions or JavaScript variables defined in the environment. This is used to detect whether the DLJS has already been loaded by Acrobat.

☛ The [*js_var*] is now optional even for users of dvipsone and dvips. If one is not provided, then appropriate code is automatically generated.

base_name is an alphabetic word with no spaces and limited to eight characters.[6] It is used to build the names of auxiliary files and to build the names of macros used by the environment.

script_name is the name of the JavaScript that you are embedding in the document. This title will appear in the document JavaScript dialog in Acrobat; unless you use Acrobat, you can't see this name in the user interface anyway. The *script_name* should be a string that is descriptive of the functionality of the code.

Commenting. Within the insDLJS environment, there are two types of comment characters: (1) a TeX comment (%) and (2) a JavaScript comment. The JavaScript comments are '//', a line comment, and '/*...*/' for more extensive commenting. These comments will survive and be placed into the PDF file. In JavaScript the '%' is used as well, use \% when you want to use the percent character in a JavaScript statement, for example app.alert("\%.2f", 3.14159);, this statement will appear within your JavaScript code as app.alert("%.2f", 3.14159);.

Example 15. The following is a minimal illustration of the use of the new environment. Here we assume the document author is using pdftex, and is not using the wonderful packages of web, exerquiz or eforms. In this case, the hyperref package with driver in the option must be introduced first, followed by insdljs with the same driver, of course. The optional argument of the insDLJS environment is not used in this example.

```
\documentclass{article}
\usepackage[pdftex]{hyperref}
\usepackage[pdftex]{insdljs}
```

[6] There is actually no limitation on the number of characters in the name, this is a legacy statement from the days of DOS, you remember DOS, don't you?

```
\newcommand\tugHello{Welcome to TUG 2001!}
\begin{insDLJS}{mydljs}{My Private DLJS}
function HelloWorld() { app.alert("\tugHello", 3); }
\end{insDLJS}
\begin{document}
\begin{Form}     % a hyperref environment, needed for \PushButton
% use built in form button of hyperref
Push \PushButton[name=myButton,onclick={HelloWorld();}]{Button}
\end{Form}
\end{document}
```

The Form environment and the \PushButton command are defined in the hyperref package. The insDLJS uses the Form environment, the eforms package defines its own \pushButton command.

Example 16. Here is the same example as above, but with dvips as the driver and using the eforms package, which calls insdljs. Note the use of the optional argument in the insDLJS environment, and the missing hyperref package statement and Form environment, the eforms package automatically inserts this code.

```
\documentclass{article}
\usepackage[dvips]{eforms}

\newcommand\tugHello{Welcome to TUG 2001!}
\begin{insDLJS}[HelloWorld]{mydljs}{My Private DLJS}
function HelloWorld() { app.alert("\tugHello", 3); }
\end{insDLJS}
\begin{document}
\pushButton[\CA{Push}\A{\JS{HelloWorld();}}]{Button}{}{11bp}
\end{document}
```

9.3. The insDLJS* Environment

The insDLJS* environment can be used to better organize, edit and debug your JavaScript. It is suitable for package developers who write a large amount of code package application.

If you have the full Acrobat product, you can open the DLJS edit dialog. There you will see a listing of all DLJS contained in the document. When you double click on one of the *script names*, you enter the edit window, where you can edit all JavaScript contained under that name.

```
\begin{insDLJS*}[js_var]{base_name}
\begin{newsegment}{script_name_1}
    ⟨JavaScript functions or exposed code⟩
\end{newsegment}
\begin{newsegment}{script_name_2}
    ⟨JavaScript functions or exposed code⟩
\end{newsegment}
...
...
\begin{newsegment}{script_name_n}
    ⟨JavaScript functions or exposed code⟩
\end{newsegment}
\end{insDLJS*}
```

Parameter Description: The environment takes two parameters, the first is optional, but required when using the Acrobat Distiller. The nested environment `newsegment` takes one required parameter.

[*js_var*] is an optional parameter, its use is discouraged.

base_name is an alphabetic word with no spaces and limited to eight characters. It is used to build the names of auxiliary files and to build the names of macros used by the environment.

script_name_i is the script name (title) that appears in the Document level JavaScript dialog of Acrobat.

9.4. Escaping

JavaScript uses the backslash as an escape character, just as does TEX. The insdljs pacakge tries to make the transition from TEX to JavaScript as easy as possible. In the table below, is a listing of the more useful characters represented by a backslash.

Sequence	Character represented
\t	horizontal tab (\u0009)
\n	newline (\u000A)
\r	carriage return (\u000D)
\"	double quote (\u0022)
\'	apostrophe or single quote (\u0027)
\\	backslash (\u005C)
\xXX	the Latin-1 character specified by the two hexadecimal digits XX
\uXXXX	the unicode character specified by the four hexadecimal digits XXXX
\XXX	the Latin-1 character specified by the octal digits XXX, between 1 and 377.

Within a JavaScript string, these special characters should be double escaped, \\, like so:

Push Me

```
\pushButton[\textFont{Arial}\CA{Push Me}\A{\JS{%
    app.alert("The \\"cost\\" of this package is \\u20AC 0.\\rThis,
    \\"\\\\\\" is a backslash");
}}]{demoEsc}{}{11bp}
```

Note the double backslash of backslash, which comes out to four, count them four backslashes, ‘’\\\\”.

Again, both JavaScript and TEX, certain punctuation marks have special meaning; in the case of JavaScript, punctuation has a special meaning within regular expressions:

Special Punctuation in Regular Expressions

```
^ $ . * + ? = ! : | \ / ( ) [ ] { }
```

When these occur in a regular expression, within a string, they need to be double escaped, \\. Outside of a sting, they need only be escaped.

Example 17. The following code searches through the string `str` and replaces every occurrence of the period character with “\.”.

```
str = "AcroTeX rocks. AcroTeX rolls."
str = str.replace(/\./g, "\\.");
```

When this code is executed, the result is “AcroTeX rocks\. AcroTeX rolls\.”.

Example 18. If one of these special characters appears outside a JavaScript string, within a regular expression pattern, for example, they need only be escaped. The code `(/\)\^\(/.test(str))` searches the string `str` for any occurrence of “)^(” and returns `true` if such a pattern is found, `false` otherwise. If `str="(x+1)^(3)"`, the search returns `true`.

9.5. Access and Debugging

For those who do not have Acrobat, the application, unless you are writing very simple code, writing and debugging JavaScript will be very difficult. From the Acrobat Reader, there is no access to the document JavaScript. You will be pretty much writing blind.

Normally, I develop the JavaScript from within Acrobat. The GUI editor does check for syntax errors, giving you a chance to correct some simple errors as you go. After I am satisfied with my code, I copy it from the editor and paste it into a `insDLJS` environment. This is how the JavaScript code of `exerquiz` was developed.

In my opinion, if you want to develop rather complicated code, having the full Acrobat product is a must. (This implies that the Windows or Mac platform is needed!)

9.6. JavaScript References

The JavaScript used by Acrobat consists of the core JavaScript plus Acrobat’s JavaScript extensions. The *Core JavaScript Reference* [2] may be found at Mozilla Developer Center. The documentation for the Acrobat extensions may be found in the guides *JavaScript for Acrobat API Reference* [4] and *Developing Acrobat Applications using JavaScript* [3], both of which I’ve had a hand in writing. These are found at Acrobat Developer Center. (Click on **JavaScript for Acrobat** in the right-hand navigation panel.)

10. Open Action

This package also defines an \OpenAction command to introduce actions that are executed when the PDF document is opened on page 1. The open action command only applies to page 1.

\OpenAction{*action_code*}

Command Location: This command must appear in the preamble of the document.

Command Description: Executes the action(s) each time page 1 is opened. The argument <action_code> is any action subtype, as listed in Section 8.5.3 of the *PDF Reference, sixth edition, PDF 1.7*. Two common types are JavaScript and Named actions. The \OpenAction command may be repeated, which will add to the list of open actions to be executed at the opening of page 1.

Special commands are defined in insdljs, \JS and \Named, that make it easy to specify these types of actions.

Example 19. \OpenAction{\JS{app.alert("Hello World!");}}

Example 20. You can use \r and \t—carriage return and tab, respectively—to format multiple lines of JavaScript:

```
\OpenAction{\JS{%
    app.alert("Hello World!");\r
    app.alert("Good Day to You!");
}}
```

Example 21. Multiple \OpenAction can be entered. Code is executed in the same order. Here, we show an alert box with a message, then jump to the last page.

```
\OpenAction{\JS{app.alert("AcroTeX rocks the world!");}}
\OpenAction{\Named{LastPage}}
```

For the other pages, beyond page 1, \thisPageAction can be used; the command can also be used for page 1 as well (it must go in the preamble).

\thisPageAction{*open_script*}{*close_script*}

Command Location: For page 1, this command must go in the preamble, otherwise, it goes on the page for which it is intended.

Command Description: *open_script* is an action that is to be executed when the current page is opened; *close_script* is an action to be executed when the current page is closed.

Example 22. Below is a simple example of how to use \thisPageAction.

```
...
\thisPageAction{\JS{console.println("Open: page 1");}}
    {\JS{console.println("Close: page 1");}}
```

```
\begin{document}
page 1
\newpage
page 2
\thisPageAction{\JS{console.println("Open: page 2");}}
    {\JS{console.println("Close: page 2");}}
...
\end{document}
```

When \thisPageAction is executed in the preamble, the <open_action> argument gets passed to the \OpenAction command.

11. The execJS Environment

This is an environment useful to PDF developers who want to tap into the power of JavaScript. To use this environment, the developer needs Acrobat 5.0 or higher. pdftex or dvipdfm can be used to produce the PDF document, but the developer needs the Acrobat product for this environment to do anything.

The execJS is used primarily for post-distillation processing (post-creation processing, in the case of pdftex and dvipdfm). The execJS environment can be used, for example, to automatically import named icons into the document, which can, in turn, be used for an animation.

The execJS is an environment in which you can write verbatim JavaScript code. This environment is a variation on insdljs, it writes a couple of auxiliary files to disk; in particular, the environment creates an .fdf file. When the newly produced PDF is loaded for the first time into the viewer (Acrobat, not Reader), the .fdf file generated by the execJS environment is imported, and the JavaScript executed. This JavaScript is *not* saved with the document. The syntax of this environment is...

```
\begin{execJS}{name}
....
⟨JavaScript code⟩
....
\end{execJS}
```

Parameter Description: The environment takes one required argument, the base name of the auxiliary files to be generated.

Many of the more useful JavaScript methods have security restrictions, the developer must create folder JavaScript that can be used to *raise the privilege* of the methods.

Example 23. Here is an extensive example taken from the AeB Pro distribution. The following code is user folder JavaScript code, see the AeB Pro documentation on how to locate the user JavaScript folder. We define a function aebTrustedFunctions that is the interface to accessing the restricted methods.

```
/*
    AEB Pro Document Assembly Methods
```

```
    Copyright (C) 2006 AcroTeX.Net
    D. P. Story
    http://www.acrotex.net
    Version 1.0
*/
if ( typeof aebTrustedFunctions == "undefined") {
    aebTrustedFunctions = app.trustedFunction(
        function ( doc, oFunction, oArgs ) {
        app.beginPriv();
            var retn = oFunction( oArgs, doc )
        app.endPriv();
        return retn;
    });
}
// Add a watermark background to a document
aebAddWatermarkFromFile = app.trustPropagatorFunction (
    function ( oArgs, doc ) {
    app.beginPriv();
        return retn = doc.addWatermarkFromFile(oArgs);
    app.endPriv();
});
```

Once this code is installed in the user JavaScript folder, and Acrobat is re-started, the code is ready to be used. The way the code is used is with the `execJS` environment.

```
\def\bgPath{"/C/acrotex/ManualBGs/Manual_AeB.pdf"}
\begin{execJS}{execjs}
    aebTrustedFunctions( this, aebAddWatermarkFromFile,
        {bOnTop: false, cDIPath: \bgPath} )
\end{execJS}
```

This is the code used to prepare this manual. It places a background graphic on each page of the document. When the newly distilled document is first opened in Acrobat, (version 7.0 or higher, is when the privilege bit started to appear), the trusted function `aebTrustedFunctions` is executed with its arguments. Looking at the definition of `aebTrustedFunctions`, what is executed is

```
app.beginPriv();
    return retn = this.addWatermarkFromFile({bOnTop: false,
        cDIPath: "/C/acrotex/ManualBGs/Manual_AeB.pdf"});
app.endPriv();
```

AeB Pro, the AcroTEX Presentation Bundle and @EASE use these execJS techniques.

12. The defineJS Environment

When you create a form element (button, text field, etc.), you sometimes want to attach JavaScript. The `defineJS` environment aids you in writing your field level JavaScript. It

too is a verbatim environment, however, this environment does not write to an auxiliary file, but saves the contents in a token register. The contents of the register are used in defining a macro that expands to the verbatim listing.

```
\begin{defineJS}[⟨tex/latex_cmds⟩]{Cmd}
...
⟨JavaScript code⟩
...
\end{defineJS}
```

Parameter Description: The defineJS environment takes two parameters, the first optional. the required parameter is the command name to be defined. Use the optional first parameter to modify the verbatim environment, as illustrated in the example below. The defineJS is a complete verbatim environment: no escape, and no comment characters are defined. You can use the optional parameter to create an escape character. You can pretty much use any character you wish, *except* the usual one '\', backslash.

Example 24. The following example illustrates the usage of the defineJS environment.

```
% Make @ the escape so we can
% demonstrate the optional parameter.
\def\HelloWorld{Hello World!}
\begin{defineJS}[\catcode`\@=0\relax]{\JSA}
var sum = 0;
for (var i = 0; i < 10; i++)
{
    sum += i;
    console.println("@HelloWorld i = " + i );
}
console.println("sum = "+sum);
\end{defineJS}
\begin{defineJS}{\JSAAE}
console.println("Enter the button area");
\end{defineJS}
\begin{defineJS}{\JSAAX}
console.println("Exiting the button area");
\end{defineJS}
\pushButton[\A {\JS{\JSA}}
    \AA{\AAMouseEnter{\JS{\JSAAE}}
        \AAMouseExit{\JS{\JSAAX}}}
]{myButton}{30bp}{15bp}
```

The code of \JSAAE and \JSAAX are so simple, the defineJS environment was really not needed. A simple \newcommand definition would have been sufficient.

See 'Inserting Complex or Lengthy JavaScript' on page 27 for an additional example of the use of the defineJS environment.

Appendices

A. The Annotation Flag F

The annotation flag F is a bit field that is common to all annotations.

Annotation Flag F

Flag	Description
\FHidden	hidden field
\FPrint	print
\FNoView	no view
\FLock	locked field (PDF 1.4)

In the user interface for Acrobat, there are four visibility attributes for a form field. The table below is a list of these, and an indication of how each visibility attribute can be attained through the F.

UI Description	Use
Visible (and printable)	
Hidden but printable	\F{\FNoView}
Visible but doesn't print	\F{\FNoPrint}
Hidden (and does not print)	\F{\FHidden}

- All fields created by the eForm commands are printable by default.

B. Annotation Field flags Ff

The table below lists some convenience macros for setting the Ff bit field.

Annotation Field flags Ff

Flag	Description	Fields
\FfReadOnly	Read only field	all
\FfRequired	Required field (Submit)	all
\FfNoExport	Used with Submit Action	all
\FfMultiline	For Multiline text field	text
\FfPassword	Password field	text
\FfNoToggleToOff	Used with Radio Buttons	Radio only
\FfRadio	Radio Button Flag	Radio if set
\FfPushButton	Push Button Flag	Push button
\FfCombo	Combo Flag	choice
\FfEdit	Edit/NoEdit	combo
\FfSort	Sort List	choice
\FfFileSelect	File Select (PDF 1.4)	text
\FfMultiSelect	multiple select (PDF 1.4)	choice
\FfDoNotSpellCheck	Do not spell check (PDF 1.4)	text, combo
\FfDoNotScroll	do not scroll (PDF 1.4)	text
\FfComb	comb field (PDF 1.5)	text
\FfRadiosInUnison	radios in unison (PDF 1.5)	radio
\FfCommitOnSelChange	commit on change (PDF 1.5)	choice
\FfRichText	rich text (PDF 1.5)	text

C. Supported Key Variables

Below is a list of the keys supported for modifying the appearance or for creating an action of a field. If the default value of a key is empty, e.g., `\Ff{}`, then that key does not appear in the widget. The Acrobat viewer may have a default when any particular key does not appear, e.g. `\W{}` will be interpreted as `\W{1}` by the viewer.

In the past, the value of the `\textColor` key must include the color model specification:

- `g` (for gray scale): a single number between 0 and 1; example, `\textColor{.5 g}`
- `rg`: Red Green Blue: a list of three numbers between 0 and 1 giving the components of color; for example `\textColor{.1 .2 .3 rg}`
- `k` Cyan Magenta Yellow [K]Black: a list of four numbers between 0 and 1 giving the components of the color according to the subtractive model used in most printers; for example `\textColor{.1 .2 .3 .4 k}`

In this current version of eforms, the color model can be optionally included. The eforms package will supply the correct specification as a function of the number of arguments provided. Thus, the examples above can now be written as `\textColor{.5}`, `\textColor{.1 .2 .3}`, and `\textColor{.1 .2 .3 .4}`.

Note: Regarding the keys `\textColor`, `\BG`, `\BC`, and `\Color`[7], beginning with eforms dated 2010/07/23 or later, eforms now uses the hycolor package to process all color keys (listed above); consequently, if the xcolor package is also loaded on your system, you can use *named colors* to specify color for the eforms keys. For example, if the definition was made

```
\definecolor{myBlue}{rgb}{0.24,0.38,0.68}
```

then each of the following is valid: `\textColor{myBlue}` (for specifying text color for text fields), `\BG{myBlue}` (for specifying the background color of a field), `\BC{myBlue}` (for specifying the border color of a field), and `\Color{myBlue}` (for specifying the border color of a link).

[7] Information regarding the `\Color` key may be found in Section 3, page 17. The `\Color` key is more fully documented in the rather comprehensive article *Support for Links in AeB/eForms* posted on the AeB Blog.

Supported Key Variables

Key	Description	Default
Entries common to all annotations:		
\F	See the annotation F flag Table	\F{}
Border Style Dictionary (BS)		
\W	Width in points around the boundary of the field, for example, \W{1}.	\W{} (same as \W{1})
\S	Line style, values are S (solid), D (dashed), B (beveled), I (inset), U (underlined); \S{B}	\S{}
\AA	Additional actions, a dictionary. These actions are triggers by mouse up, mouse down, mouse enter, mouse exit, on focus, on blur events; for text and editable combo boxes there is also the format, keystroke, validate and calculate events. The various triggers are discussed in Trigger Events.	\AA{} (no actions)
\A	Action dictionary, use this to define JavaScript actions, as well as other actions, for mouse up events. See Trigger Events for a discussion of the mouse up event.	\A{} (no action)
\Border	Used with link annotations, an array of three numbers and an optional dash array. If all three numbers are 0, no border is drawn	\Border{0 0 0} (no border)
\AP	Appearance dictionary, used mostly in AcroTeX with check boxes to define the 'On' value.	\AP{}
\AS	Appearance state, normally used with check boxes and radio buttons when there are more than one appearance. Advanced techniques only.	\AS{}
Entries common to all fields:		
\TU	Tool tip (PDF 1.3), for example, \TU{Address}. Obeys unicode option.	\TU{}

Key	Description	Default
\Ff	See the Field flag Ff table; e.g. \Ff{\FfReadOnly} makes the field read only.	\Ff{}
\DV	Default value of a field. This is the value that appears when the field is reset; e.g., \DV{Name:}. Obeys unicode option.	\DV{}
\V	Current value of the field; for example, \V{D. P. Story}. Obeys unicode option.	\V{}

Entries specific to a widget annotation:

\H	Highlight, used in button fields and link annotations. Possible values are N (None), P (Push), O (Outline), I (Invert); e.g., \H{P}.	\H{} (same as \H{I})

Appearance Characteristics Dictionary (MK)

\MK	A dictionary that contains the keys listed below. For all fields the MK has a template that is filled in using the keys below; this key is available only for check boxes and radio buttons.	various
\R	Number of degrees the field is to be rotated counterclockwise. Must be a multiple of 90 degrees; \R{90}.	\R{}
\BC	The boundary color, a list of 0 (transparent), 1 (gray), 3 (RGB) or 4 (CMYK) numbers between 0 and 1. For example, \BC{1 0 0} is a red border.	\BC{} (transparent)
\BG	Background color. Color specification same as \BC	\BG{} (transparent)

Key	Description	Default
\CA	Button fields (push, check, radio) The widget's normal caption; e.g. \CA{Push}, in the case of a push button. For check boxes and radio, the value of \CA is a code that indicates whether a check, circle, square, star, etc. is used. These codes are introduced through \symbolchoice. Obeys unicode option.	\CA{}
\RC	Push button fields only. The roll over text caption. Obeys unicode option.	\RC{}
\AC	Push button fields only. The down button caption. Obeys unicode option.	\AC{}
\mkIns	A variable for introducing into the MK dictionary any other key-value pairs not listed above. Principle examples are I, RI, IX, IF, TP, which are used for displaying icons on a button field. See an example in the demo file eforms.tex	\mkIns{}
\I	(push buttons only) an indirect reference to a form XObject defining the buttons's *normal icon*	\I{nIcon}
\RI	(push buttons only) an indirect reference to a form XObject defining the buttons's *rollover icon*	\RI{rIcon}
\IX	(push buttons only) an indirect reference to a form XObject defining the buttons's *down icon*	\I{dIcon}
\TP	(push buttons only; optional) A code indicating the layout of the text and icon; these codes are 0 (label only); 1 (icon only); 2 (label below icon); 3 (label above icon); 4 (label to the right of icon); 5 (label to the left of icon); 6 (label overlaid on the icon). The default is 0.	\TP{1}

Key	Description	Default
\SW	(push buttons only; optional) The *scale when key*. Permissible values are A (always scale), B (scale when icon is too big), S (scale when icon is too small), N (never scale). The default is A.	\SW{A}
\ST	(push buttons only; optional) The *scaling type*. Permissible values are A (anamorphic scaling); P (proportional scaling). The default is P.	\ST{P}
\PA	(push buttons only; optional) The *position array*. An array of two numbers, each between 0 and 1 indicating the fraction of left-over space to allocate at the left and bottom of the annotation rectangle. The two numbers should be separated by a space. The default value, \PA{.5 .5}, centers the icon in the rectangle.	\PA{0 0}
\FB	(push buttons only; optional) The *fit bounds* Boolean. If true, the button appearance is scaled to fit fully within the bounds of the annotation without taking into consideration the line width of the border. The default is false.	\FB{true}

Entries common to fields containing variable text:

\Q	Quadding for text fields. Values are 0 (left-justified), 1 (centered), 2 (right-justified); e.g., \Q{1}.	Q{} (left justified)

Default Appearance (DA)

\DA	Default appearance string of the text in the widget. Normally, you just specify text font, size and color. Can be redefined, advance techniques needed.	
\textFont	Font to be used to display the text	\textFont{Helv}
\textSize	size in points of the text	\textSize{9}

Key	Description	Default
\textColor	color of the text, there are several color spaces, including grayscale and RGB; for example, \textColor{1 0 0 rg}, gives a red font. Recent advances in parsing this command have eliminated the need to include the color space specification. Thus, \textColor{1 0 0} also gives a red font.	\textColor{0 g}
Entries specific to text fields:		
\MaxLen	The maximum length of the text string input into a text field. Used also with comb fields to set the number of combs. Example, \MaxLeng{15}.	\MaxLen{}
Entries specific to signature fields:		
\Lock	This key is used to lock fields after the signature field is signed. Example, \Lock{/Actions/All}. See subsection 2.4, page 14 for more examples.	\Lock{}
Specialized, non-PDF Spec, commands:		
\rawPDF	If all else fails, you can always introduce key-value pairs through this variable.	\rawPDF{}
\autoCenter	There is a centering code that attempts to give a pleasant placement of the field. Say \autoCenter{n} to turn this off.	
\presets	This commands takes a macro as its argument, the text of the macro are key-value pairs. This is useful for setting up a series of presets for fields. Example, \presets{\myFavFive}	

Key	Description	Default
\symbolchoice	Use this variable to specify what symbol is to be used with a check box or radio button. Possible values are check, circle, cross, diamond, square and star. Can be used to globally change the symbol choice as well; for example, \symbolchoice{check}, which is the default value.	
\linktxtcolor	The value of this variable is a named color and is the color of the link text. Only recognized in link annotations. A value of \linktxtcolor{} paints the text the \normalcolor.	\linktxtcolor {\@linkcolor}

References

[1] *Core JavaScript Guide*, available from Mozilla Developer Center. See page 25.

[2] *Core JavaScript Reference* available from Mozilla Developer Center. See pages 25 and 48.

[3] *Developing Acrobat Applications using JavaScript*, available from Acrobat Developer Center See pages 25 and 48.

[4] *JavaScript for Acrobat API Reference*,available from Acrobat Developer Center See pages 25 and 48.

[5] *PDF Reference, sixth edition, PDF 1.7*, available from Acrobat Developer Center See pages 21, 22, 23, and 24.