Techniques of Introducing Document-level JavaScript into a PDF file from a LaTeX Source

D. P. Story
Department of Mathematics and Computer Science, University of Akron, Akron, OH 44325
dpstory@uakron.edu
http://www.math.uakron.edu/~dpstory/

Abstract

The method of introducing Acrobat document-level JavaScript (DLJS) into a PDF depends on the application used: pdftex or dvipdfm. Until recently, users of Acrobat's distiller did not have the ability to automatically introduce such JavaScript into the document from a LaTeX source. For users of Acrobat 5.0, this situation is, at last, rectified. The focus of this paper is to enumerate, illustrate and discuss these various methods.

A new package, `insDLJS`, is also introduced. This package enables both package and document authors to insert document-level JavaScript.

Introduction

Adobe's Portable Document Format (PDF) has become widely used by some as a document exchange format; LaTeX users routinely convert their research papers and technical documents to PDF and post them on the Internet, or e-mail them to a interested colleague. When the LaTeX package `hyperref`, written by Sebastian Rathz and Heiko Oberdiek, is used, cross-references created by standard LaTeX commands are automatically converted into hyperlinks within the PDF document.

However, LaTeX and PDF are capable of more than just producing an electronic copy of a paper document. By utilizing LaTeX's package customization feature (LaTeX packages) and Acrobat's form elements and powerful JavaScript language, an author can build a colorful, richly interactive PDF. Such a document can be used as a teaching tool that would quicken the interest of the student.

The focus of this paper is on JavaScript. JavaScript can be attached to a PDF document in many ways, for example, to a form button. JavaScript attached to buttons can be used to initiate actions. Any scripts that are repeatedly used, are general (not form specific), or are rather lengthy, can be placed at what Acrobat calls the *document-level*; they are called document-level JavaScripts (DLJS). A button action, then, can simply call these DLJS to perform various calculations or tasks.

For authors or package writers who really want to tap into the power of JavaScript, document-level scripts need to be written that perform the target task and embedded into the PDF. In the context of developing an educational document, many revisions to the document are typically made and the JavaScript must be reinserted each time the PDF is built. Ideally, the JavaScripts should appear within the LaTeX source file, which makes it easy to access and modify them, and then *automatically* inserted when the PDF is built.

The applications pdftex and dvipdfm both support the insertion of DLJS from a LaTeX source. Historically, it was not possible to insert DLJS from a LaTeX source through the distiller method, and sadly, *this is still the case*. However, the Acrobat application, version 5.0, does include some extensions that can be exploited to obtain an automatic insertion of DLJS. For version 5.0, it is the Acrobat application that inserts the DLJS automatically following distillation, *not* the distiller itself.

The discussion that follows—for the case of production of PDF by the distiller method—assumes that the author has the full Acrobat 5.0 product. It is important to note that it is only necessary for the *author* to have full Acrobat 5.0; the user, the one who reads the document, needs only have the Acrobat Reader (version 4.0 or later).

Inserting DLJS

We begin by discussing the basic techniques for each of the common applications used for producing a PDF document. Suppose we wish to introduce the following JavaScript function at the document level.

```
function HelloWorld()
{
    app.alert("Hello World!", 3);
}
```

A variation on a rather well-known, yet useless function.

In the sections that follow, it is assumed the source document uses the `hyperref` package, with the appropriate driver setting: `dvips`, `dvipsone`, `pdftex` and `dvipdfm`. Some of the commands in the code below have their definitions in the `hyperref` package.

Within a package, or preamble, make the following definitions. This code is common to all cases, pdftex, dvipdfm and distiller.

```
\def\jsR{\string\r}  % carriage return in JS
\def\jsT{\string\t}  % tab in JS
\begingroup\obeyspaces\obeylines%
\global\let^^M=\jsR%
\gdef\DLJS{%
function HelloWorld()
{
    app.alert("Hello World!", 3);
}
}%
\endgroup
```

This defines a macro that expands to the code of the desired JavaScript function. Note that `\obeyspaces` and `\obeylines` are used; `\global\let^^M=\jsR` will insert a `\r` at the end of each line of the JavaScript code, this nicely formats the script within the PDF file itself.

pdftex and dvipdfm The approach to DLJS insertion is similar in the case of these two applications, both of which fully support, through primitives or specials, DLJS insertion.

In each case, insertion is a two step process: (1) create an PDF object dictionary containing the necessary key-value pairs

```
<< /S /JavaScript /JS (\DLJS) >>
```

and (2) enter the script name and reference offset into the `Names` array of the `JavaScript` dictionary.

The implementation of these two steps is different for these two applications.

We wrap the code in the next subsections in a `\AtBeginDocument` command. For private macros, this is not necessary, but for public macro packages it is useful to delay the insertion of the code until after the preamble.

pdftex For pdftex, the following is all the additional code needed to get DLJS insertion:

```
\AtBeginDocument
{%
  \immediate\pdfobj
    { << /S /JavaScript /JS (\DLJS) >> }
  \xdef\objDLJS{\the\pdflastobj\space 0 R}
  \immediate\pdfobj {%
    << /Names [(Doc Level JS) \objDLJS] >> }
  \edef\objNames{\the\pdflastobj\space 0 R}
  \pdfnames {/JavaScript \objNames}
}
```

The control sequences `\pdfobj` and `\pdfnames` are pdftex primitives.

dvipdfm Similarly, for dvipdfm, we need only include the following code to get the DLJS:

```
\AtBeginDocument{%
  \immediate\@pdfm@mark{obj @objDLJS
    << /S /JavaScript /JS (\DLJS) >> }
  \immediate\@pdfm@mark{obj @objnames
    << /Names [(Doc Level JS)  @objDLJS] >> }
  \@pdfm@mark{put @names
    << /JavaScript @objnames >> }
}
```

The command `\@pdfm@mark` is defined in terms of a dvipdfm `\special`,

```
\def\@pdfm@mark#1{\special{pdf: #1}}
```

as defined in `hyperref`.

Multiple Functions More than one function can be grouped together using the obvious approach:

```
\begingroup\obeyspaces\obeylines%
\global\let^^M=\jsR%
\gdef\DLJS{%
function HelloWorld()
{
    app.alert("Hello World!", 3);
}
function GetNumberofPages()
{
    app.alert("The number of pages is "
        + this.numPages);
}
}%
\endgroup
```

Now, we proceed as described above.

An alternate approach is to have separate definitions for each of you functions, say, `\DLJSi` and `\DLJSii`, and insert each of them separately into the JavaScript dictionary. For example, in the case of `pdftex`, we could include the following code:

```
\AtBeginDocument
{%
  \immediate\pdfobj
    { << /S /JavaScript /JS (\DLJSi) >> }
  \xdef\objDLJSi{\the\pdflastobj\space 0 R}
  \immediate\pdfobj
    { << /S /JavaScript /JS (\DLJSii) >> }
  \xdef\objDLJSii{\the\pdflastobj\space 0 R}
  \immediate\pdfobj {%
    << /Names
        [
            (Doc Level JS) \objDLJSi\space
            (More DLJS)    \objDLJSii
        ]
```

```
    >> }
  \edef\objNames{\the\pdflastobj\space 0 R}
  \pdfnames {/JavaScript \objNames}
}
```

Similarly for dvipdfm.

Acrobat expects the script names of the DLJS (these are '`Doc Level JS`' and '`More DLJS`', in the example above) to be *sorted*; otherwise, the Acrobat application does not give editing access to the scripts that are 'out-of-sorts'.[1] The scripts would be accessible from within Acrobat or Reader, but would not necessarily be available for editing within Acrobat.

Acrobat 5.0 For authors that use Acrobat 5.0 (those that use dvips or dvipsone to produce a PostScript file and then distill) the problem is a little more complicated. Version 5.0 comes with an extended FDF (Forms Data Format) specification. This new specification allows DLJS to be placed within the FDF file. The FDF file is then imported into the document and the DLJS is inserted.

The Concepts Take our basic `HelloWorld` JavaScript function, and place it into the FDF file, which I'll call `dljs.fdf`, as follows:

```
%FDF-1.2
1 0 obj
<<
  /FDF
  <<
    /JavaScript
    <<
      /Doc 2 0 R
    >>
  >>
>>
endobj
2 0 obj
[ (Doc Level JS)  3 0 R ]
endobj
3 0 obj
<<>>
stream
function HelloWorld()
{
  app.alert("Hello World!",3);
}
endstream
endobj
trailer << /Root 1 0 R >>
%%EOF
```

Once this file has been created, it can be inserted into the PDF file by including the following code in your LaTeX document:

```
\AtBeginDocument{\literalps@out{%
[ {Page1} << /AA << /O << /S /JavaScript /JS
 (%
  if(typeof HelloWorld == "undefined")\jsR\jsT
  this.importAnFDF("dljs.fdf");
  )
 >> >> >>
/PUT pdfmark}}
```

This code creates an open page action for page 1 of the PDF. When the document is first opened in the Acrobat application, usually immediately following distillation, if the function `HelloWorld` is not already defined, the JavaScript method `importAnFDF` will import the specified FDF into the document; otherwise, the open action does nothing. The JavaScript function `HelloWorld` will be placed at the document-level because of the structure of the FDF file.

A Simple Implementation The FDF can be created and edited as a separate file; however, our goal was to have all code contained in the LaTeX source file itself. A simple solution would be to use a verbatim write to write the necessary code to a FDF file.

For example, consider the following code. It is assumed that the `verbatim` package has been loaded.

```
\makeatletter
\newwrite \dljs@FDF

% open a stream
\immediate\openout\dljs@FDF=dljs.fdf
\let\verbatim@out=\dljs@FDF

% verbatimwrite Environment: Writes to current
%\verbatim@out. Based on examples and macros
% of the verbatim package. Set \verbatim@out
% before calling this macro
\newenvironment{verbatimwrite}
 {\@bsphack
  \let\do\@makeother\dospecials
  \catcode`\^^M\active
  \def\verbatim@processline{%
    \immediate\write\verbatim@out
        {\the\verbatim@line}}%
  \verbatim@start}%
 {\immediate\closeout\verbatim@out\@esphack}
```

Now write the FDF file from the LaTeX source.

```
\begin{verbatimwrite}
%FDF-1.2
1 0 obj
<<
  /FDF
  <<
    /JavaScript
    <<
```

[1] Neither pdftex or dvipdfm sorts the `Names` array by script names.

```
      /Doc 2 0 R
    >>
  >>
>>
endobj
2 0 obj
[ (Doc Level JS) 3 0 R ]
endobj
3 0 obj
<<
>>
stream
function HelloWorld()
{
    app.alert("Hello World!", 3);
}
endstream
endobj
trailer
<<
  /Root 1 0 R
>>
%EOF
\end{verbatimwrite}
```

Use `pdfmark` to import the FDF when Acrobat is first opened.

```
\AtBeginDocument{\literalps@out{%
[ {Page1} << /AA << /O << /S /JavaScript /JS
 (%
  if(typeof HelloWorld == "undefined")\jsR\jsT
  this.importAnFDF("dljs.fdf");
  )
 >> >> >>
/PUT pdfmark}}
\makeatother
```

Once the DLJS has been inserted, save the file (using "Save As"). The DLJS now is saved with the file. When the document is opened in Acrobat (or Reader) the JavaScript is available to be executed. You can delete the open action at this point.

Multiple Functions Multiple functions can either be written to the same FDF file, or to separate FDF files. When you import multiple FDF files into Acrobat using `importAnFDF`, their script names are automatically sorted, see the footnote at the end of 'Multiple Functions', page 162.

Plain TEX Users The above examples illustrate the basics of DLJS inclusion. Plain TEX users can (easily) adapt these techniques to plain TEX. This is left as an exercise.

The `insDLJS` Package

All the code described earlier works very well, and is adequate for someone trying to develop materials for the WWW or for a CD-ROM using a private macro package. Such a person typically uses only one system—pdftex, dvipdfm, or the distiller—to create the materials. These macros cannot be used conveniently as part of a package, where the user—the one using the package—may want to modify the script and/or define a custom script. A general insert DLJS package is needed. In this section, we discuss a new package called `insDLJS`.

The package `insDLJS` takes the basic techniques already discussed, modifies them so that the insertion will be friendly to package authors who use `insDLJS` to write LATEX packages, and to document authors who want to jazz up their documents with DLJS as well.

The package has four driver options: `dvipsone`, `dvips`, `pdftex` and `dvipdfm`. The `insDLJS` also defines a new environment called `insDLJS`; the syntax is

```
\begin{insDLJS}[<func>]{<base>}{<name>}
<JavaScript functions or exposed code>
...
...
\end{insDLJS}
```

where,

`#1:` This optional parameter, `<func>`, is *required* for the `dvipsone` and `dvips` options; otherwise it is ignored. It's value must be the name of one of the functions defined in the environment.

`#2:` This parameter, `<base>`, is an alphabetic word with no spaces. It is used to build the names of auxiliary files and to build the names of macros used by the environment.

`#3:` The third parameter, `<name>`, is the script name of your JavaScript. This name appears in the "JavaScript Functions" dialog of Acrobat, under the menu

```
Tools > JavaScript > Document JavaScripts..
```

This environment writes one file (`<base>.def`), or possibly two files (`<base>.def` and `<base>.fdf`).

In the case of `pdftex` or `dvipdfm` options, the `<base>.def`, which contains the necessary definitions and code, as discussed in 'pdftex', page 162 and the following section on 'dvipdfm', is read back into the output document to set the DLJS code.

In the distiller case, `<base>.def` is read in and second file, `<base>.fdf`, is written. It is the file `<base>.fdf` that is imported into Acrobat following distillation, as outlined on page 163.

The DLJS defined within the `insDLJS` environment in a package and in a preamble of a document will be included in the PDF document automatically. It is important to choose a base name, `<base>`, and a script name, `<name>`, *different* from any that has

already been declared earlier; otherwise, code will be overwritten or simply not appear.

Example 1. The following example assumes the driver is either `dvipsone` or `dvips`; in these cases the first parameter is required.

```
\begin{insDLJS}[HelloWorld]{mypkg}{Pkg DLJS}
function HelloWorld()
{
    app.alert("Hello World!", 3);
}
\end{insDLJS}
```

This would result in the automatic insertion of the specified JavaScript at the document-level when the PDF is built.

In this case, the `importAnFDF` method (JavaScript) is used, as discussed on page 163. The value of the optional argument, '`HelloWorld`', is used to detect whether the script has already been imported. The environment generates the following code:

```
\AtBeginDocument{\literalps@out{%
[ {Page1} << /AA << /O << /S /JavaScript /JS
 (%
  if (typeof HelloWorld == "undefined")\jsR\jsT
  this.importAnFDF("mypkg.fdf");
  )
 >> >> >>
/PUT pdfmark}}
```

The optional argument is ignored and need not even be included when `pdftex` or `dvipdfm` is specified as a package option. □

Example 2. A LaTeX package writer may want to use the `insDLJS` environment to insert DLJS into the package; also, a document author who uses such a package may want to define *additional* DLJS. In this case, just place the code in the preamble, for example,

```
\begin{insDLJS}{dljs}{Private DLJS}
function tugWelcome()
{
    app.alert("Welcome to TUG 2001!", 3);
}
\end{insDLJS}
```

The script will be automatically introduced into the PDF document. □

The `Exerquiz` Package[2] has been rewritten to use the `insDLJS` package. This not only makes the JavaScript code easier to read, modify and maintain, it also allows users of `Exerquiz` to add in their own DLJS. `Exerquiz` uses the base name of `exerquiz`.

Example 3. A package author who uses the package `insDLJS` to insert DLJS may wish the package user to modify the scripts in some authorized way. For example, the package author may include

```
\newcommand\Hello{"Hello World!"}
\begin{insDLJS}
function HelloWorld()
{
    app.alert(@Hello, 3);
}
\end{insDLJS}
```

The script included in the output file (`.dvi` or `.pdf`) using the `\AtBeginDocument` mechanism. The user then has the opportunity to redefine `\Hello` to, say,

```
\renewcommand\Hello{"Bonjour le Monde\space !"}
```

Now the `HelloWorld` function will create an alert dialog that says, "Bonjour le Monde !". □

Important. Note that '`@`' is used as the escape character within the environment. The `insDLJS` environment eventually leads to a `verbatim` environment. The at-char, '`@`', has its catcode changed, `\catcode`\@=0`; this will allow macros to expand within the `verbatim` environment. The backslash '`\`' is needed in JavaScript since it is used as an escape. JavaScript does *not* use the '`@`' symbol at all, so we are free to use it here.

A Complete Example The following listing is a complete example of usage of the `insDLJS` package.

```
\documentclass{article}
\usepackage[pdftex]{hyperref}
\usepackage[pdftex]{insdljs}
\newcommand\tugHello{Welcome to TUG 2001!}
\begin{insDLJS}{mydljs}{Private DLJS}
function HelloWorld()
{
    app.alert("@tugHello", 3);
}
\end{insDLJS}
\begin{document}
\begin{Form}     % needed for \PushButton
\section{Test of the \texttt{insDLJS} Package}
% use built-in form button from hyperref
Push \PushButton[name=myButton,
        onclick={HelloWorld();}]{Button}
\end{Form}
\end{document}
```

A Double-Edged Problem In the process of extending the `exerquiz` package to include the ability to evaluate objective-style questions using DLJS, one big problem was encountered; consider the following example:

```
\begin{insDLJS}[<func>]{<base>}{<name>}
function TestForLParen(string)
{
    if (/\(/.test(string))
        app.alert("Found Left Parenthesis!",3);
}
\end{insDLJS}
```

[2] CTAN:macros/latex/contrib/supported/webeq

The Problem. The above script will search the parameter `string` for a left parenthesis, '`(`'. When dvipsone or dvips is used (i.e., the distiller is used), the `<base>.fdf` file is created, imported into the PDF file and the script works correctly. When pdftex or dvipdfm is used this code does not work! When the file is opened in Acrobat an exception is thrown, and an error message appears:

```
SyntaxError: unterminated parenthetical (
```

In the case of pdftex and dvipdfm, the JavaScript code is written directly to the PDF file, bypassing distiller. The problem with the above code turns out to be *unbalanced parentheses*. (Acrobat treats '`\(`' as a left parenthesis, not an escaped special character.) All special characters, therefore, need to be escaped; we want

```
\begin{insDLJS}[<func>]{<base>}{<name>}
function TestForLParen(string)
{
    if (/\\\(/.test(string))
        app.alert("Found Left Parenthesis!",3);
}
\end{insDLJS}
```

This code now works when the option `pdftex` or `dvipdfm` is used; however, when this code is used with the distiller (with the `dvipsone` or `dvips` option), *Acrobat 5.0 crashes*!

The Solution. We have two opposing problems, solving one creates another. The solution is to introduce a command, `\e`. (The '`e`' is for escape.) The macro has two definitions: for the `dvipsone` and `dvips` options, we define `\gdef\e{}`; for `pdftex` and `dvipdfm` options, we make the following definitions:

```
\catcode`\@=0 @catcode`@\=12 @gdef@e{\}
```

Note that in all cases, we do change the catcode of '`@`' to `\catcode`\@=0`.

Thus, to get the function `TestForLParen` to be properly defined for all options, we must type:

```
\begin{insDLJS}[<func>]{<base>}{<name>}
function TestForLParen(string)
{
    if (/@e\@e(/.test(string))
        app.alert("Found Left Parenthesis!",3);
}
\end{insDLJS}
```

This problem is not limited to regular expressions. Anywhere there is an escape character '`\`', there is a potential for problems. For example, if you want to assign the variable `x` a value of '`\`', you would have to say, within some `insDLJS` environment:

```
var x = "@e\@e\";
app.alert(x, 3);
```

In systems using the `dvipsone/dvips`, this would expand to `var x = "\\";` in contrast, on systems using the `pdftex/dvipdfm` option, it would expand to `var x = "\\\\";`. Tricky!

Here is a final complete example.

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

\newcommand\tugHello{Welcome to TUG 2001!}

\begin{insDLJS}{mydljs}{My DLJS}
function HelloWorld()
{
    app.alert("@tugHello", 3);
}
function TestForLParen(string)
{
    if (/@e\@e(/.test(string))
        app.alert("Found Left Parenthesis!",3);
}
\end{insDLJS}

\begin{document}
\begin{Form}

\section{Test of the \texttt{insDLJS} Package}

\begin{itemize}
    \item \PushButton[name=myButton,
        onclick={HelloWorld();}]{Button:}
    \item \TextField
            [name=myText,width=2in,
             keystroke={if(event.willCommit)
                TestForLParen(event.value);}
            ]{Text Field:}
\end{itemize}

\end{Form}
\end{document}
```

A button and a text field are created. Click on the button to get an alert dialog. Enter some text in the field; when the data is committed, the function `TestForLParen` is called, and the field is scanned for a left parenthesis.

A Note to pdftex and dvipdfm Users If you plan to make extensive use of DLJS, and to write complex functions, such as the ones that appear in the exerquiz package, the purchase of the full Acrobat suite is *strongly* recommended.

The only way to debug Acrobat JavaScript is through the JavaScript edit window of the Acrobat viewer. If you only use Acrobat Reader, there is very little feedback as to what goes wrong with a particular script; development and debugging of scripts will be a very long and tedious process.

All the JavaScripts in the exerquiz package were first written from within the JavaScript editor and

tested. Once the script was operating correctly, it would be copied, and pasted into the exerquiz source file for automatic inclusion.

Final Comments

Acrobat Reader provides a forms capability and a powerful JavaScript engine that can be exploited to create dynamic, interactive PDF documents. The LaTeX system with its flexible "plug-in" capability (LaTeX packages) is a solid authoring tool for creating high-quality typeset content. The LaTeX packages that give access to PDF are

1. The `hyperref`[3] package written by Sebastian Rathz, Heiko Oberdiek, *et al.*, automatically creates bookmarks and cross-reference hyperlinks, allows document information fill-in and provides basic form creation code, to mention a few. This package is fundamental to any author who wants to create PDF documents for distribution over the Web.
2. The `webeq`[4] (D. P. Story) and `pdfscreen`[5] (Radhakrishnan CV) packages are screen oriented packages. With them, an author can develop a document with page size suitable for presentations or on-line viewing. These packages come with other bells and whistles as well.
3. The `exerquiz`[6] package (D. P. Story) defines several environments for creating exercises and quizzes, both multiple choice and fill-in. This package makes extensive use of the form features and (document-level) JavaScript.
4. The `insDLJS` package gives the package or document author an easy and convenient method of including document-level JavaScripts.

There are many other packages (e.g., for graphics insertion, color creation) too numerous to mention.

As a result of the above mentioned packages, as well as the various applications for producing PDF (pdftex, dvipdfm and the distiller), an author now has a fairly complete set of authoring tools for developing such a colorful, visually attractive and highly interactive documents.

[3] CTAN:macros/latex/contrib/supported/hyperref
[4] CTAN:macros/latex/contrib/supported/webeq
[5] CTAN:macros/latex/contrib/supported/pdfscreen
[6] CTAN:macros/latex/contrib/supported/webeq