AcroTEX.Net

The AeB Pro Package
Attachments and Doc Assembly

D. P. Story

Prepared: May 9, 2007

http://www.acrotex.net
Presented: January 12, 2007

Table of Contents

1. Introduction

AeB Pro has two options for attaching files to the source PDF. The approach is the `importDataObject` JavaScript method in conjunction with the FDF techniques.

Also in this sample file, doc assembly techniques are also demonstrated.[1]

2. Attaching Files with AeB Pro

There are two options for attaching files

1. `attachsource` is a simplified option for attaching any file of the form `\jobname.`*ext*.
2. `attachments` is a general option for attaching a file, as specified by its absolute or relative path.

2.1. The `attachsource` option

Use this option to attach a file with the same base name as `\jobname`.

```
\usepackage[%
    driver=dvips,
    web={
        pro,
        ...
        usesf
    },
    attachsource={tex,dvi,log,tex.log},
    ...
```

[1] The attachments feature uses doc assembly methods, but simplified for user convenience.

```
]{aeb_pro}
```

Simply list the extensions you wish to attach to the current document. In the example above, we attach the original source file `\jobname.tex`, `\jobname.dvi`, `\jobname.log` (the distiller log) and `\jobname.tex.log` (the tex log).

Important: There should be no space following a comma in the lists of extensions. Thus, the list should be

```
attachsource={tex,dvi,log,tex.log}
```

not

```
attachsource={tex, dvi, log, tex.log}
```

or

```
attachsource={tex,
    dvi,
    log,
    tex.log
}
```

However, the following works,

```
attachsource={
    tex,%
    dvi,%
    log,%
    tex.log
}
```

Frankly, the argument list for extensions is so short, there is no reason to put them on separate lines.

One problem with attaching the log file is that the distiller also produces a log file with the same name `\jobname.log`. Consequently, the log file for the tex file is overwritten by the distiller log file. You'll see from the PDF document, that the log file attached is the one for the distiller.

A work around for this is to latex your file, rename the log file to another extension, such as `\jobname.tex.log`, then distill.

2.2. The `attachments` option

The `attachments` key is for attaching files other than ones associated with the source file. The value of this key is a comma-delimited list (enclosed in braces) of absolute paths and/or relative paths to the file required to attach. For example,

```
\usepackage[%
    driver=dvips,
    web={
        pro,
        ...
        usesf
    },
    attachments={robot man/robot_man.pdf,%
    /C/Documents and Settings/dps/My Documents/My Pictures/birthday17.jpg},
    ...
]{aeb_pro}
```

The first reference is relative to the folder that this source file is contained in (and is attached to this PDF), and second one is an example of an absolute path (picture not attached).

There are some files that Acrobat does not attach, but there is no public list of these. One finds them by discovery, `.exe` and `.zip` files, for example.

A trick that I use to send `.zip` files through the email (they are often stripped away by mail servers) is to *hide* the `.zip` file in a PDF as an attachment. But since Acrobat does not attach `.zip`, I change the extension from `.zip` to `.txt`, then inform the recipient to save the `.txt` file and change the extension back to `.zip`. Swave!

3. Doc Assembly

Ahhhh, document assembly. What can be said? This is a method that I have used for many years and is incorporated into the insdljs package under the name of `execJS`. Whereas the `execJS` environment is still available to you, I've simplified things. The term doc assembly refers to the use of the `docassembly` environment (which is just an `execJS` environment).

The `execJS`/`docassembly` environments create an FDF file with the various JavaScript commands that were contained in the body of the environment. These environments also place in open page action so that when the PDF is opened for the first time in Acrobat Pro, the FDF file will be imported and the JS will be executed one time and then discarded, see [1] for an article on this topic. This technique only works if you have Acrobat Pro.

In addition to the `docassembly` environment, AeB Pro also has several macros that expand to JavaScript methods that I find useful. These are

1. `\addWatermarkFromFile`: inserts a background into the PDF

2. \importIcon: imports icon files[2]
3. \importSound: imports a sound file
4. \appopenDoc: opens a document
5. \insertPages: inserts pages into the PDF, useful for inserting pages of difference sizes, such as tables or figures, into a LaTeX document which requires that all page be of a fixed size.
6. \importDataObject: Attaches a file to the PDF. This function is used in the two attachments options of AeB Pro.

See the AeB Pro documentation for details. Here, in this demo file, I present the code in the preamble of this document:

```
\begin{docassembly}
\addWatermarkFromFile({
    bOnTop:false,
    cDIPath:"/C/AcroTeX/AcroPackages/ManualBGs/Manual_BG_DesignV_AeB.pdf"
});
\end{docassembly}
```

It is *very important* to note that the arguments for this (pseudo-JS method) are enclosed in matching parentheses/braces combination, i.e., ({...}). The arguments are key-value pairs separated by a colon, and the parameters themselves are separated by commas. (The argument is actually an object-literal). It is *extremely important* to have the left parenthesis/brace pair, ({, immediately follow the function name. This is because the environment is a partial-verbatim

[2] The AcroMemory package uses these environments and functions to import icons.

environment: \ is still the escape, but left and right braces have been "sanitized". The commands, like \addWatermarkFromFile first gobble up the next two tokens, and re-inserts ({ in a different location. (See the aeb_pro.dtx for the definitions.)

For another cheesy demonstration, let's import a sound, associate it with a button. I leave it to you to press the button at your discretion.


```
\begin{docassembly}
try {
    \addWatermarkFromFile({
        bOnTop:false,
        cDIPath:"/C/AcroTeX/AcroPackages/ManualBGs/Manual_BG_DesignV_AeB.pdf"
    });
} catch(e) { console.println(e.toString()) };
try {
    \importSound({cName: "StarTrek", cDIPath: "../extras/trek.wav" });
} catch(e) { console.println(e.toString()) };
\end{docassembly}
```

Above is the full verbatim listing of the docassembly environment that will execute for the screen. You'll note the \importSound command, which imports the sound file trek.wav. I've also enclosed the individual commands in a try/catch construct. Doing so is very useful for debugging the script. Also, in the case of the background file Manual_BG_DesignV_AeB.pdf, which is not distributed, an exception will be raised when you try to compile this file. The try/catch will catch the exception, and send an error message back to the console, but allows processing to continue; consequently, you should get the sound file imported and the above

button should work.

One last little demonstration of the doc assembly methods. In the preamble, I've imported a few AeB logos (forgive me) and placed them as appearance faces for the button above. Below is a listing of the code, with some comments added.

```
\begin{docassembly}
...
...
// Import the sounds into the document
\importIcon({cName: "logo", cDIPath: "../extras/AeB_Logo.pdf"});
\importIcon({cName: "logopush", cDIPath: "../extras/AeB_Logo_bw15.pdf"});
\importIcon({cName: "logorollover", cDIPath: "../extras/AeB_Logo_bw50.pdf"});
var f = this.getField("cheesySound");           // get the field object of the button
f.buttonPosition = position.iconOnly;           // set it to receive icon appearances
var oIcon = this.getIcon("logo");               // get the "logo" icon
f.buttonSetIcon(oIcon,0);                       // assign it as the default appearance
oIcon = this.getIcon("logopush");               // get the "logopush" icon
f.buttonSetIcon(oIcon,1);                       // assign it as the down appearance
oIcon = this.getIcon("logorollover");           // get the "logorollover" icon
f.buttonSetIcon(oIcon,2);                       // assign it as the rollover appearance
\end{docassembly}
```

The result is the button you see above.

As a final example of `docassembly` usage, rather than using the attachments options of AeB Pro, you can also attach your own files using the `docassembly` environment.

```
\begin{docassembly}
...
...
```

```
try {
    \importDataObject({cName: "AeB Pro Example #2",cDIPath: "aebpro_ex2.pdf"});
} catch(e){}
\end{docassembly}
```

The attachments options automatically assign names. These names appear in the Description column of the attachments tab of Acrobat/Reader. For file attached using the `attachsource`, the base name plus extension is used, for the files specified by the `attachments` key, the names are given sequentially, `"AeB Attachment 1"`, `"AeB Attachment 2"` and so on. When you roll your own, the description can be more aptly chosen.

I have found many uses for the `execJS` environment, or the simplified `docassembly` environment. You are only limited by your imagination, and knowledge of JavaScript for Acrobat.

References

[1] "execJS: A new technique for introducing discardable JavaScript into a PDF from a LaTeX source," TUGboat, The Communications of the TeX User Group, Vol. 22, No. 4, pp. 265-268 (2001). 6