**PhUSE 2015**

**Paper TS06**

# Unleash the power of Adobe Acrobat using JavaScript

Oliver Wirtz, Bayer Pharma AG, Wuppertal, Germany

## ABSTRACT

Results of statistical programming are usually presented in portable document format (PDF). Such files normally consist of more than one table listing or figure and can become really big (several thousand pages). Therefore, features like bookmarks, links and table of contents (TOC) are needed to help users browsing the document. Adobe Acrobat® is a software often used to create merged PDF files but advanced features like TOC need to be added to PDFs in separate workflows or require additional software packages. This paper will show how to use Adobe Acrobat´s built-in JavaScript® capabilities to manipulate PDF documents.

## INTRODUCTION

Electronic report documents are almost always formatted as Acrobat Portable Documents Files (PDF). They can consist of several thousand pages and therefore need to have features like bookmarks, links and table of content (TOC) to enable reviewers to electronically browse the document. Such document features are usually added in a separate workflow and use metadata information generated at some point during the initial production of result outputs. In any case, they are part of the clinical SAS programming workflow. Initial setup and maintenance of metadata driven approaches is complex and often not worth the effort for smaller short-term projects like ad-hoc analyses. Adobe Acrobat is a software widely used in the pharmaceutical industry to manually create and manipulate PDF files. This can involve merging multiple PDF files, adding or deleting pages, creating overall pagination of the document and many more. Albeit its popularity hardly anybody knows that tasks in Adobe Acrobat can be automated using the built-in Acrobat JavaScript (AcroJS) engine. Acrobat JavaScript is an extension of core JavaScript version 1.5 and allows to enhance Acrobat functionalities without the need to use a high level programming language such as C or C++. The AcroJS object model is well documented online [1] and code examples help to understand how certain objects work. The following sections will show the use of basic AcroJS commands to develop a real-live application that adds a table of content (TOC) to a PDF created from multiple PDF files. The application was developed in Acrobat Standard XI but also tested in Acrobat Standard 7.

## PREPARATION OF ACROBAT

Before starting code development in Acrobat we need to make sure that JavaScript is activated. In order to do that, open Acrobat, go to Edit -> Preferences and make sure that “Enable Acrobat Java Script” is enabled in category JavaScript. Our script will be an application level script placed in a folder location that Acrobat scans on startup. Acrobat has a public and a personal folder location for JavaScripts:

In Windows® systems the public folder can be found under programs (C:\Program Files\Adobe\Acrobat 11.0\Acrobat\Javascripts). Code snippets placed in this location are compiled when Acrobat is started. However, the public folder is not suitable for code development and normally users do not have write access to it. On the other hand, the public folder is useful for deployment of finalized applications especially when Acrobat is hosted on a server. For code development the personal script folder should be used. In Acrobat versions older than 10.1.1 it can be found under C:\Users\<Username>\Application Data\Adobe\Acrobat\XX.X\JavaScripts. Newer Acrobat versions use a different location (C:\Users\<Username>\AppData\Roaming\Adobe\Acrobat\Privileged\XX.X\JavaScripts) which needs to be created manually.

### A VERY FIRST CODE EXAMPLE

Acrobat provides a JavaScript debugger that can be used to test shorter code snippets. In Acrobat Pro X and XI the debugger can be started from the toolbar menu (Tools -> JavaScript debugger). In other versions (e.g. Acrobat Standard XI) the debugger cannot be accessed via toolbar buttons. In order to open it you need to add a custom menu item. Place the following line of code into a text file with the *.js extension and place it into your personal JavaScript folder.

```
app.addMenuItem({cName:"Console Window", cParent:"View", cExec:"console.show()"});
```

After (re-)starting Acrobat you should see a new menu item called “Console Window” under “View”. The debugger is not further discussed in this paper but it may be useful to test code snippets in your own projects. The use of the debugger is explained in detail under [2].

## CREATING A TABLE OF CONTENTS

With JavaScript activated and scripting folders in place we can now start to take a look at the actual project. Let´s assume we want to create a PDF document that consists of two tables and a listing. The overall plan is to manually combine multiple PDF files into one document and to use AcroJS to read out the bookmark information to create a TOC. In addition the script will be used to add header information and overall pagination. The fully documented source code can be found on GitHub [7].

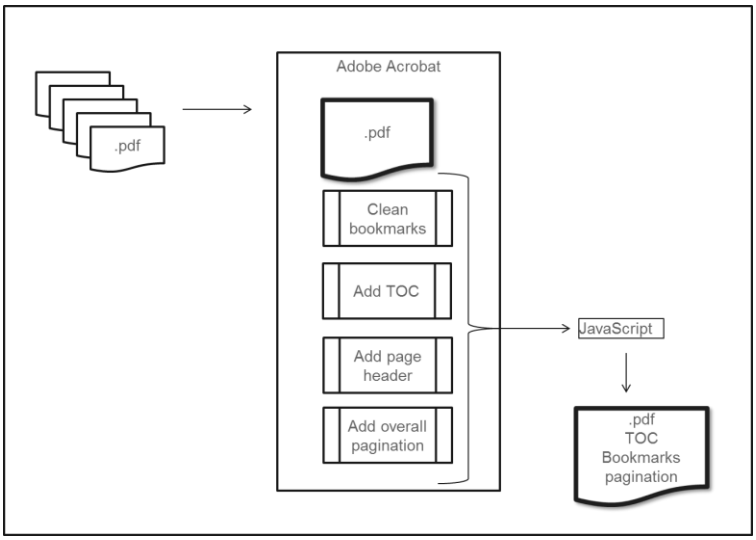



**Figure 1: Summary of tasks**

### STEP 1: COMBINING PDF FILES

The very first step is to manually create a PDF from single PDF files. The sample PDFs used in this project were initially created in SAS using SAS ODS PDF with the ODS PROCLABEL statement to directly produce outputs with bookmarks. Unfortunately, ODS PROCLABEL does not only produce the bookmark we want to see in the combined document. It also generates additional child bookmarks like in figure 1 below.

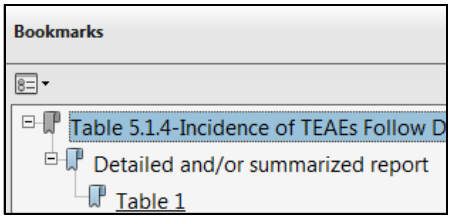



**Figure 2: Bookmark structure using ODS PROCLABEL**

But even if source PDF documents already have the desired bookmark structure there is some re-work needed after single files were combined into one document. The reason is that Acrobat does not simply merge the files but also produces a new parent bookmark consisting of the filename. Unfortunately, this “feature” cannot be disabled. After merging documents the bookmark structure would look like in figure 3 below.

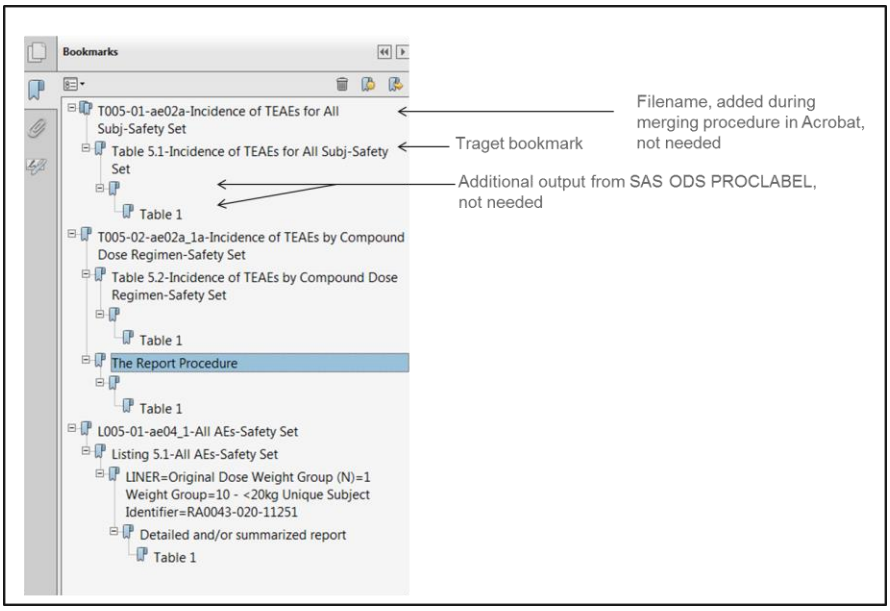
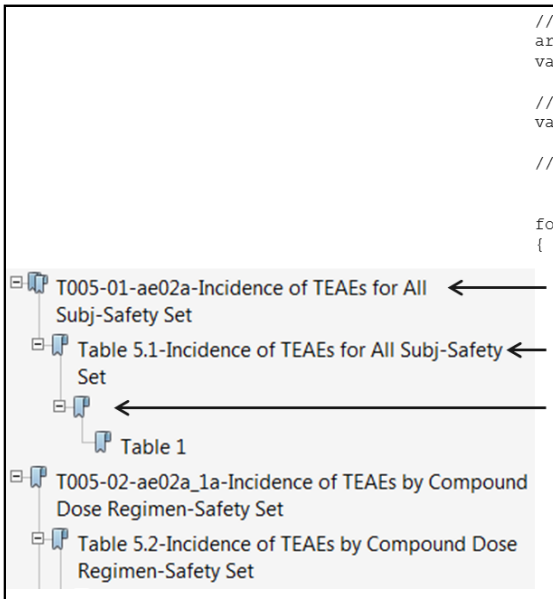



**Figure 3: Bookmark structure after combining files**

**STEP 2: DELETE UNWANTED BOOKMARKS**

After the first step to manually produce a combined PDF we now want to use AcroJS to delete all unwanted bookmarks. The bookmark object in AcroJS allows to scan the complete bookmark tree visible in the bookmark pane. As you can see from figure 3 the target bookmark we want to keep is always a child bookmark of the filename bookmark created during the merging process. In order to access the target bookmark we need to go to the respective parent bookmark and then one level deeper to the first child bookmark. In AcroJS this would look like this:




```
//this is kind of the master bookmark, the one where all the others
are attached to
var bm = this.bookmarkRoot;

//count the number of bookmarks, here the filename bookmarks
var ibmLength = bm.children.length;

//loop through all filename bookmarks;


for (var i = 0; i < ibmLength; i++)
{
    // Check if bookmark[i] contains sub-ordinate bookmarks
    var bmToCheck = bm.children[i];


     if((bmToCheck.children!=null))
     {
   //check if these again have child bookmarks
      if(bmToCheck.children[0].children!=null)
      {
       //count child bookmarks;
        var mycount=bmToCheck.children[0].children.length;
             for(var k=0; k<mycount; k++)
            {
            subBmToDeleteArray[k]=bmToCheck.children[0].children[k];
                 for(var k=0; k<mycount; k++)
                         {
                         subBmToDeleteArray[k].remove();
                         }
            }
//move table bookmark to end of tree;
       bm.insertChild(bmToCheck.children[0], bm.children.length);

// Add parent bookmark (filename bookmark) to another array of
bookmarks to be deleted later;
       bmToDeleteArray[bmToDeleteArray.length] = bmToCheck;
    }
}
```

**Figure 4: Code snippet to demonstrate use of the AcroJS bookmark object, example taken from [4]**

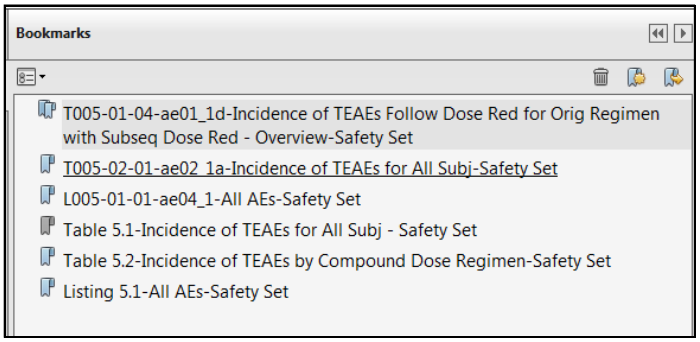



**Figure 5: Table bookmarks moved and additional ODS PROCLABEL bookmarks deleted**

Executing this first part of the code deletes the child bookmarks introduced by ODS PROCLABEL and moves the original table bookmarks to the same level as filename bookmarks (figure 5). In the next step the filename bookmarks will be removed. Since these were already stored as bookmark objects in array `bmToDeleteArray` they can be deleted easily by parsing the array:

```
for (var i = 0; i < bmToDeleteArray.length; i++)
{
   bmToDeleteArray[i].remove();
}
```

Figure 6 below shows the final appearance of the document. Only bookmarks needed to browse the document are left. These elements will be used in the next step to setup the TOC pages.

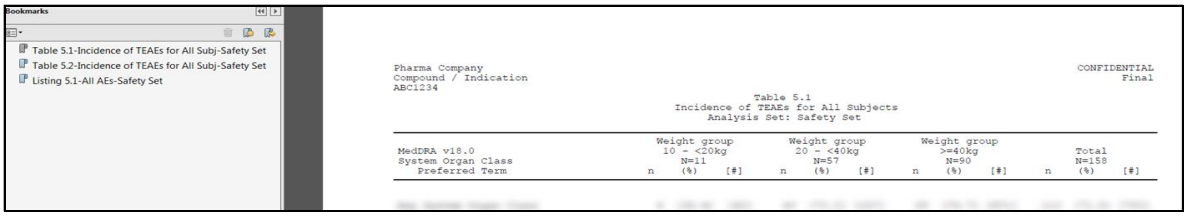



**Figure 6: Bookmark structure after cleaning**

#### STEP 3: EXTRACT HEADER DATA

We want the appearance of the TOC to match the main document as close as possible. That means, we also want to see the common page header on the TOC pages. Many elements are static and can be hard coded. But apart from company name, confidentiality statement or final/draft status there are variable elements like compound name, indication or study codes that change from study to study. In our project we want to copy this information from the first page of the document and use it in the TOC. Unfortunately, in AcroJS does not scan words from top of the page. Pages in PDF are setup using coordinates and the origin of a page starts with 0,0 in the lower left corner of a page. So rather than simply looking for the static content like “CONFIDENTIAL” in the standardized document header, we need to scan the complete document starting in the footer of the page. In figure 7 below the word “CONFIDENTIAL” was found at position 95 and the word “Final” at position 99. We now want to extract all words between these two words to get compound and indication. The study code is the first word following “FINAL” and will be extracted as such.

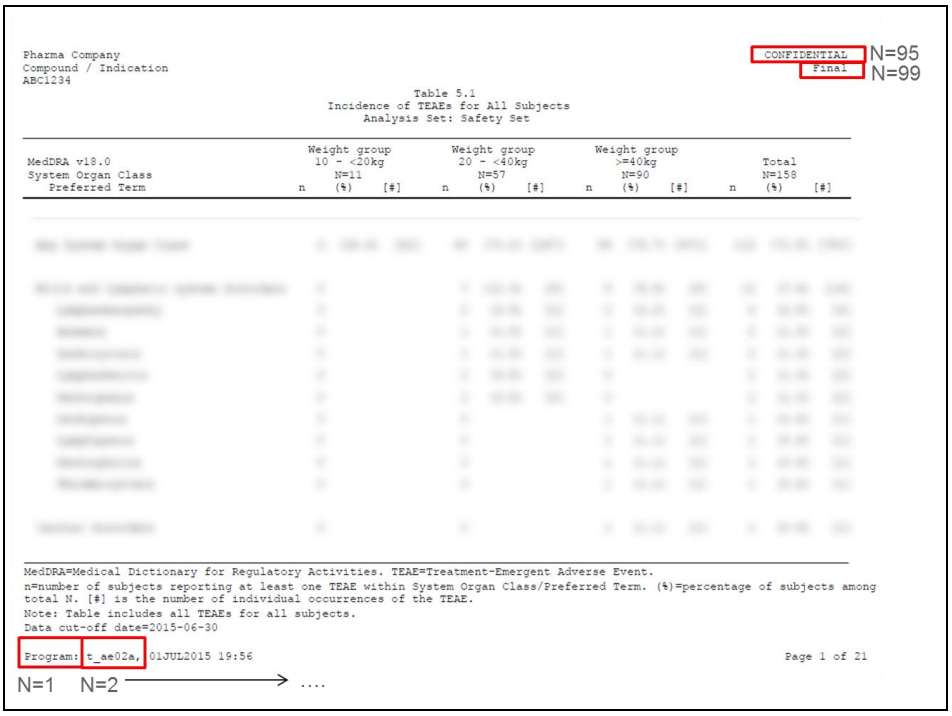



**Figure 7: Direction of word scan starts in lower left corner, target index in header is depending on length of words in footer**

Fortunately, we need to scan all words of the first page only. You will note that all indices in Acrobat start with 0. Page 1 of a document has index 0, the first word on a page is also indexed 0. Therefore we count all words on page 1 using

```
numWords = this.getPageNumWords(0);
```

and after that loop through all words and to retrieve the position of static header items:

```
for (var j = 0; j < numWords; j++)
  {
   ckWord = this.getPageNthWord(0, j);
   if (ckWord =="CONFIDENTIAL"){k=j}
   if (ckWord == "Draft" || ckWord == "Final") {l=j;}
```

when both items were found, read out words between k and l and concatenate them. After that, read in the study code that is the first word after “Final”:

```
   if ((k>=0) && (l>0))
     {
     for (var h=k ; h <= l+1; h++)
       {projectnam = projectnam.trim() + " " + this.getPageNthWord(0, h, false);
        projectnam = projectnam.trim();
       }
   if (h==l )
     {
       studynam = this.getPageNthWord(0, h+1);
       studynam = studynam.trim();
     }
   }
```

**STEP 4: SETUP TOC PAGES**

Now we need to determine the total number of TOC pages needed. Using Courier New in size 9 allows for 16 entries to fit on one page in landscape format. A line can be up to 125 characters long and to fit on the printable part of the page. The bookmark object we want to use for the TOC content cannot keep more than 127 characters, so there is a limitation, anyway.
Let´s calculate the total number of TOC pages from the total number of bookmarks using the bookmarksroot object.

```
var tocpages=Math.ceil(bm.children.length/16);
```

Now that we know the number of TOC pages we can insert the pages with header information included. Acrobat adds new pages to the end of the document. The new pages therefore need to be moved to the beginning of the document.

```
for (var m = 0; m < tocpages; m++)
       {
          this.newPage({nWidth:792, nHeight:612});
          this.movePage(this.numPages-1,-1);

          var f = this.addField("header1", "text", 0,[61,550,90,535]);
              f.textColor=color.black;
              f.textSize=9;
              f.textFont=font.Cour;
              f.strokeColor= color.transparent;
              f.fillColor= color.transparent;
              f.value="PharmaCompany";
              f.alignment="left";


[…]
```

The code above shows how the first bit of header text (“PharmaCompany”) is populated on the empty page. Unfortunately, there is no explicit “write a text” object in AcroJS. In order to populate text on the page we are using the field object that is normally used to create text fields in forms. The field object is added using the addField method. This method assigns a name (“header1”) and the type (“text”), and also defines the page and the location of the field on the page. This technique will also be used to create the other text elements of the TOC as well as the overall pagination.
After producing blank TOC pages with header information we can now add the TOC entries:

**STEP 5: CREATING TOC ENTRIES**

This step is rather simple. We want to create TOC entries from bookmarks. Using the this.bm.bookmark object we can loop through all child bookmarks and read out the bookmark text. The destination page of the bookmark cannot be read from the bookmark object. So, in order to retrieve the destination page we programmatically execute the bookmark and then read-in the page number of the current page.

```
var bmtext = this.bookmarkRoot.children[i].name ;
this.bookmarkRoot.children[i].execute();
var TOCpagenum = this.pageNum + 1;
```

Remember that page indices start with 0 in AcroJS. Therefore we need to add 1 to the result in order to get the correct “physical” PDF page number. Let´s say the bookmark has a length of 52 characters and we want to have one blank after the bookmark and one blank before the page number. When the total length of the string is 125 characters we need to fill up the space between the bookmark string and the page number with dots.

```
bmtext = bmtext + " ";
while (bmtext.length < 122)
   {
     bmtext = bmtext.concat(".");
     bmlen = bmlen + 1;
   }
bmtext = bmtext.concat(" ",TOCpagenum );
```

The TOC entry is then populated using the addField method as before. The very first TOC entry would look like this

```
var f = this.addField(TOC_txt1, "text", 0, [61, 580, 790, 610]);
    f.textColor=color.blue;
    f.textSize=9;
    f.alignment="left";
    f.textFont=font.Cour;
    f.strokeColor= color.transparent;
    f.fillColor= color.transparent;
    f.value= bmtext;
```

Finally the TOC page looks like the example below.

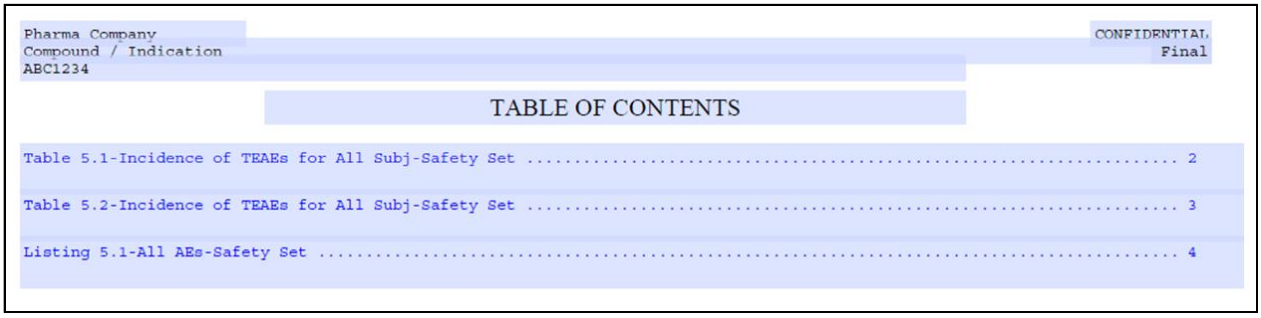



**Figure 8: TOC appearance after adding texts as form fields**

The appearance is not optimal yet. The blue background color of TOC elements indicates that we had to use a form field with some default text as value. Fortunately, there is a method to remove all “extras” from objects and make form fields appear as normal text.

```
this.flattenPages()
```

removes all additional functionality from the form field and makes it part of the document content. Figure 9 shows the appearance of the TOC after flattening. The TOC is now available as text but we still need to add links.

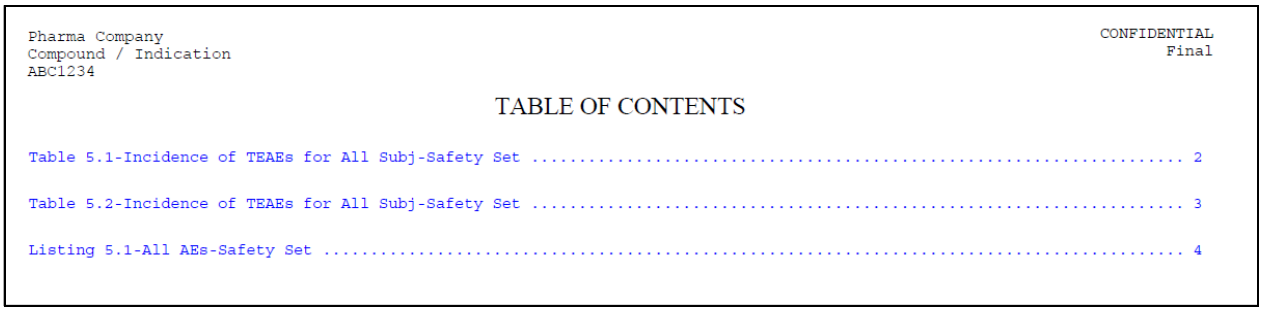



**Figure 9: TOC after applying this.flattenPages()**

#### STEP 6: ADDING LINKS

The syntax for link elements is similar to form fields. In general it defines a rectangle that links to the destination page. We just need to re-iterate through all bookmarks and use the bookmark destination in the link object. The location of the link is using the same coordinates as the TOC entry:

```
var mylink = this.addLink(0, [61,550,90,535]);
```

```
mylink.setAction("this.pageNum = " + this.pageNum);
```

In general we would combine adding TOC text and links at the same time. However, using the flattenPages method not only converts form field to text but also removes all link items from the document. Therefore we need to add all text elements first, flatten the document and then add all link elements in a separate step.
After adding links, the TOC is done and the document is ready to use.

#### STEP 7: SETUP IN ACROBAT

With all code in place we want to be able to call the program as easy as possible. In order to do that, all code is wrapped into a trusted function call to keep the code backward s compatible for Acrobat versions earlier than 10.1.1.

```
MakeTOC= app.trustedFunction( function ()
{
app.beginPriv();


…place all the code here…


 app.endPriv();

 });
```

Finally, the code should be available to users within Acrobat. A new menu item can be added using

```
app.addMenuItem({ cName: "MakeTOC", cParent: "File",
                  cExec: "MakeTOC();", nPos: 0});
```

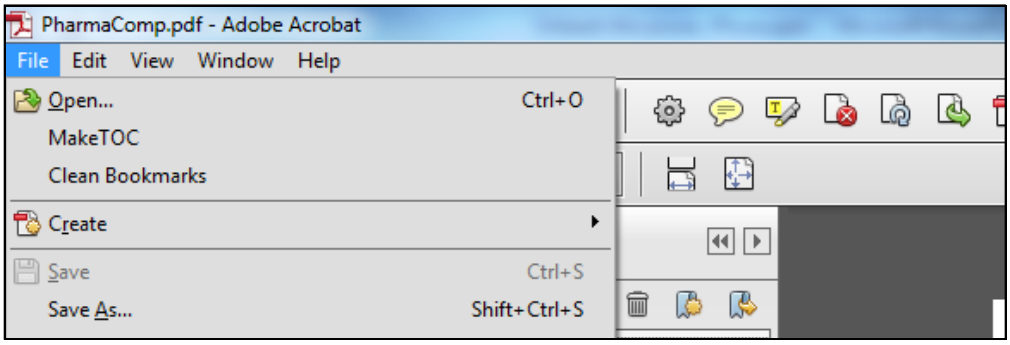



**Figure 8: New items added to the common file menu**

## DISCUSSION

Although PDF documents are compiled and processed using the Acrobat software the scripting capability with AcroJS is unknown to most users in the pharma industry. This paper showed the general setup of Adobe Acrobat to be able to use AcroJS as well as a couple of useful scripting methods to demonstrate how AcroJS can help with routine work. The application discussed here can be fully customized and standardized to company needs. It does not involve any manual editing and therefore allows creating PDF bundles fast and accurate. Especially in daily routine work and during internal review cycles this is a very simple way to update merged documents. However, using JavaScript to create TOC links introduces JavaScript code into the actual PDF document. This can be problematic when documents need to go into Document Management Systems (DMS). Many DMS applications are still compatible with Acrobat version 5 only which in turn is not compatible to JavaScript. Moreover, the FDA does not accept documents using JavaScript [5].
Therefore, the (basic) methods discussed above should be used in documents needed for internal review or prior to submission activities. The author does not see a real issue here because about 98% of daily business is related to internal customers and will not be submitted to external parties like the FDA. Submission compliant documents, however, can also be created using AcroJS but require advanced knowledge about the use of Acrobat Distiller® and

PostScript®. A code example utilizing PostScript elements is also available on GitHub [6]. More details about the use of PostScript in PDF were published previously and can be found in [7]. The most important methods and their use are documented in the Adobe online help, other functionality, i.e. not supported officially can be found in specialized user forums only.

## CONCLUSION

In general, programming in AcroJS is straightforward and should encourage SAS programmers to investigate AcroJS as an additional option to post-process PDF documents. AcroJS can also be used to simply extend the basic functionality of the Acrobat Software for free. Once developed, code can be deployed and maintained easily and even non-programmers can use it when program calls are added as menu items.

## REFERENCES

[1] Acrobat API reference:
http://help.adobe.com/livedocs/acrobat_sdk/10/Acrobat10_HTMLHelp/wwhelp/wwhimpl/common/html/wwhelp.htm?context=Acrobat10_SDK_HTMLHelp&file=JS_Dev_Overview.71.1.html (accessed on 27AUG2015)

[2] Use of Acrobat Debugger:
https://acrobatusers.com/tutorials/javascript_console (accessed on 27AUG2015)

[3] Lawhorn, Barri: Let´s Give `Em Something to TOC About: Transforming the Table of Contents of Your PDF File, SESUG 2011 http://analytics.ncsu.edu/sesug/2011/SS03.Lawhorn.pdf

[4]
http://www.planetpdf.com/developer/article.asp?ContentID=RemovingfilenamebookmarkscreatedbyAcrobat&gid=6881 (accessed on 27AUG2015)

[5] Portable Document Format (PDF) Specification:
http://www.fda.gov/downloads/Drugs/DevelopmentApprovalProcess/FormsSubmissionRequirements/ElectronicSubmissions/UCM163565.pdf (accessed on 27AUG2015)

[6] https://github.com/OliverWirtz/acro_toc (accessed on 27AUG2015)

[7] Merging RTF files using SAS®, MSWord®, and Acrobat Distiller®
Pharmaceutical Programming (December 2012), pp. 50-56

## ACKNOWLEDGMENTS

The author wants to thank Sandra Wirtz, Ankur Mathur and Sascha Ahrweiler for reviewing the manuscript.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Oliver Wirtz
Bayer Pharma AG
Aprather Weg
D-42113 Wuppertal
Deutschland
Email: oliver.wirtz@bayer.com