

Batch Sequences:

Tips, Tricks and Examples

Introduction

This document discusses fundamentally two topics: (1) basic concepts and techniques of using the “Execute JavaScript” sequence of the Batch Processing feature of Acrobat 5.0; (2) proposed solutions to some of the common problems users often encounter and seek simple solutions to. (See [Problems](#) section below.)

All batch sequence files discussed in this article are available from the Acrobat 5.0 CD. To use them, copy them into the user *Sequences* folder. (If necessary, search for the word “Sequences” using *Start > Find > Files and Folders* to locate this folder.) When you start the Acrobat viewer the next time, Acrobat will read this folder and list the sequences under the *File > Batch Processing* menu. Read this article in its entirety before attempting to use these sequences.

Definitions

Selected Files

A batch sequence performs repetitive operations on a collection (or selection) of files. In the “Batch Edit Sequence” dialog, there is a “Run commands on” item. A dropdown list offers four choices

- Selected Files
- Selected Folders
- Ask When Sequence is Run
- Files Open in Acrobat

In this document, the term “selected files” refers to the files chosen through this dropdown list.

Basic Concepts

All JavaScripts are executed as a result of a particular event, and batch processing is no different. When an “Execute JavaScript” batch sequence is run, a “Batch Event” occurs and an event object is created. During the event, the script is repeatedly executed on each of the [Selected Files](#).

When writing scripts for a batch sequences, a developer needs to be aware of the [Event Object](#) and its properties and the [this Object](#). A brief explanation of these follows.

Event Object

target

The `target` property of the event object during a batch sequence points to the *Doc object* of the file currently being processed. Thus, for example,

```
event.target.path
```

is the device independent path of the file currently being processed.

rc

Setting `rc` (the return code) to *false* will abort the batch process, no other files, if any, will be processed; however, `event.rc = false` does not abort the script, the rest of the code lines in the program flow will be executed. One strategy to avoid the continuation of the flow is to throw an exception, and set `event.rc = false` with the catch:

```
try {
    ... JS script for batch processing .....
    if ( somethingGoesWrong ) throw "Aborting Process"
    ....JS script for batch processing ....
} catch (e) {
    ... clean up code lines ....
    event.rc = false;
}
```

this Object

The *this* object also points to this *Doc object* of the file currently being processed in batch. Thus, for example, the script

```
var annots = this.getAnnots({nSortBy: ANSB_Author});
```

gathers all annotations in the document (currently being processed), sorts them by author, and returns an *annot object* for further manipulation.

Global Variables

Variables that are needed to hold values across document processing need to be declared as global.

```
global.report = new Report()
global.counter = 0;
```

Now the *global.report* object is available for each of the [Selected Files](#). Within the script that processes a file we might have, for example,

```
global.counter++
```

```
global.report.writeText("Report on File \#" + global.counter);
```

At the end of the batch job, any global variables can then be removed; for example,

```
delete global.counter;
```

Begin/End Batch Job

More complicated batch jobs, ones that involve cross-document reporting, may require some start up code, **Begin Job** code, in which certain variables are initialized (using [Global Variables](#)) before processing begins. The files are then processed, after which time some additional code, **End Job** code, might be required to clean up or write a report. The batch feature of Acrobat 5.0 does not, however, have a built in capability for **Begin Job** and **End Job** code, yet, a workable system can be devised.

Programmatically detecting the beginning of the job is simple enough. A global variable, such as, *global.counter*, can be used to detect the beginning of the job as well as to count the files as they are processed. Initially, the variable *global.counter* is undefined, when the batch is run, *global.counter* is initialized to zero. The code below illustrates how *global.counter* is used.

Acrobat does not signal that the current document being processed is, in fact, the last one; therefore, to create an **End Job** code segment, you must know programmatically what file is the last one. One simple device is to count the number of PDF files to be processed. The *global.counter* can then be used to determine when the current file is the last file. See the code below.

Counting the number of files to be processed can be accomplished with a utility batch sequence, such as the one called [Count PDF Files](#) given below. **Count PDF files** defines a global variable *global.FileCnt*, and sets the value of this global to the number of files to be processed. (Of course the [Selected Files](#) must be the same as the files you intend to process using your JavaScript code.)

Below is a general outline of how **Begin Job** code and **End Job** code can be inserted using the ideas discussed above.

```
// Begin Job
if ( typeof global.counter == "undefined" ) {
    console.println("Begin Job Code");
    global.counter = 0;
    // insert beginJob code here
    .....
}
// Main Code to process each of the selected files
try {
    global.counter++
    console.println("Processing File #" + global.counter);
    // insert batch code here.
```

```

    .....
}catch(e) {
    console.println("Batch aborted on run #" + global.counter);
    delete global.counter;    // so we can try again, and avoid End Job code
    event.rc = false;        // abort batch
}

// End Job
if ( global.counter == global.FileCnt ) {
    console.println("End Job Code");
    // insert endJob code here
    .....
    // may have to remove any global variables used in case user wants to run
    // another batch sequence using the same variables, for example...
    delete global.counter;
}

```

Debugging and Testing

The batch feature is a very useful feature for performing a number of tasks on large scale; but, it can also be very dangerous to use. Any automated utility that saves files to a hard drive can potentially create havoc on that drive.

Here are some debugging and testing tips:

- When testing the script, use test documents, rather than “company” documents.
- Initially, work with a small number of test documents.
- Use test directories which contain files that you don’t mind being overwritten.
- Make sure your paths exist. If an *app.openDoc* is executed with a path that does not exist, an exception is thrown.
- Use try/catch/throw to exit gracefully from a batch if something goes wrong (such as bad paths), as well as to better control the flow of the code. See the code snippet in the section discussing [event.rc](#).
- Use *console.println()* to write information to the console to give you feedback as to the value of different variables you are using. Always assume a batch is not doing what you think it is doing. The *console.println()* can be used to peek into the mind of the sequence as it is running.
- When potentially executing an infinite loop, again, during the testing phase, limit the loop to a definite finite number of executions until you are sure the code is correct. See the [Populate and Save](#) code.
- Code that uses Begin Job/End Job depends on certain global variables being undefined at the start of the batch. Make sure these variables are undefined to insure the script will run correctly.

Problems

In this section, we discuss a number of problems Acrobat users have sought simple solutions to for a long time. Problems are divided into different categories: bookmarks, icons, database and so on.

Some solutions are fairly general, others specific to the system on which they were developed: specific paths to folders are specified, specific icons used, etc. Individuals using these sequences should study them carefully, and modify them to suit the need and the system.

Important: Though fully tested, these scripts are offered with no guarantees. All scripts discussed in this article are available on the Acrobat 5.0 CD.

Alphabetical Listing

Below is an alphabetical listing of all sequences discussed in this article, all of which come on the Acrobat CD.

- [Change Security to None.sequ](#)
- [Copy Bookmarks from Array.sequ](#)
- [Copy Bookmarks from Doc.sequ](#)
- [Count Bookmarks.sequ](#)
- [Count PDF Files.sequ](#)
- [Cross Doc Comment Summary.sequ](#)
- [Extract Pages to Folder.sequ](#)
- [Gather Bookmarks to Array.sequ](#)
- [Gather Bookmarks to Doc.sequ](#)
- [Gather DigSig Info.sequ](#)
- [Import Named Icons.sequ](#)
- [Insert Barcode.sequ](#)
- [Insert Bookmarked Pages.sequ](#)
- [Insert Navigation Icons.sequ](#)
- [Insert Stamp.sequ](#)
- [List all Bookmarks.sequ](#)
- [Populate and Save.sequ](#)
- [Signature Sign All.sequ](#)
- [Spell Check a Document.sequ](#)

Utility Sequences

The following sequences are used by some of the problem oriented sequences.

Count PDF Files

```
/* Count PDF files */
if ( typeof global.FileCnt == "undefined" ) global.FileCnt = 0;
global.FileCnt++
```

This sequence sets the global variable, *global.FileCnt*, equal to the number of [Selected Files](#). If used prior to running another batch sequence that depends on this value, be sure the [Selected Files](#) are the same.

Gather Bookmarks to Array

This sequence gathers up bookmark information from the [Selected Files](#) and saves it in a global array. The information in this array is used by the sequence [Copy Bookmarks from Array](#), explained below.

```
/* Gather Bookmarks into an Array */

// Begin Job Setup
if ( typeof global.bmArray == "undefined" ) {
    global.indexCnt = 0;
    global.bmFileCnt = 0;
    global.bmArray = new Array();
}

global.bmFileCnt++;                // count files while we're at it

// Get the path of the file Acrobat has taken from the selected folder
var path2file = this.path;

// regular expression acquire the base name of file
var re = /\.*\|\.pdf$/ig;

// filename is the base name of the file Acrobat is working on
var filename = path2file.replace(re,"");

// If title is available, use it for a bookmark, else use filename
if ( (this.info.Title == null) || (this.info.Title.replace(/\s/g,"") == "" ) )
    var bmToFile = filename;
else
    var bmToFile = this.info.Title;

global.bmArray[global.indexCnt] = [bmToFile,filename];

global.indexCnt++;
```

Solution Notes: Once the array *bmArray* is in memory, you can sort the entries with respect to some criteria. The following code would alphabetically sort the bookmark names. The script could be run from the console, or you can insert it into the “Begin Job” section of [Copy Bookmarks from Array](#).

```
function compare (a,b) {                // define a compare function
```

```

        if (a[0] < b[0] ) return -1;
        if (a[0] > b[0] ) return 1;
        return 0;
    }
    global.bmArray.sort(compare);

```

Bookmarks

There are many problems associated with bookmarks that can be easily solved using the batch sequencing capability of Acrobat.

List all Bookmarks

Problem: Create a PDF document that lists the bookmarks in another PDF document. The former document can be saved and/or printed.

Solution:

```

/* List all Bookmarks */
/* Recursively work through bookmark tree */
function PrintBookmarks(bm, nLevel)
{
    if (nLevel != 0) { // don't print the root
        bmReport.absIndent=bmTab*(nLevel-1);
        bmReport.writeText(util.printf("%s",bm.name));
    }
    if (bm.children != null)
        for (var i = 0; i < bm.children.length; i++)
            PrintBookmarks(bm.children[i], nLevel + 1);
}

// set up parameters to write a report
bmTab = 20;
bmReport = new Report();
bmReport.size = 2; // large font size for title
bmReport.writeText(this.title);
bmReport.writeText(" "); // skip a line
bmReport.size = 1.5; // slightly smaller for heading
bmReport.writeText("Listing of Bookmarks");
bmReport.writeText(" ");
bmReport.size = 1; // default size for everything else
PrintBookmarks(this.bookmarkRoot, 0); // start moving through bookmarks

// make it global so the object will be "remembered" after the batch is done.
global.bmRep = bmReport; // make global for next step

/* We now want to open our report (Report.open), but we cannot open or save
   (Report.save) a report while a modal dialog box is open, and there is on

```

```

while batch running. So, we'll wait until batch is done, then we'll open
the report. This step could be a problem child, if not set up correctly.
*/
global.wrtDoc = app.setInterval(
    'try {'
    +'    reportDoc = global.bmRep.open("Listing of Bookmarks");'
    +'    console.println("Executed Report.open");'
    +'    app.clearInterval(global.wrtDoc);'
    +'    delete global.wrtDoc;'
    +'    console.println("Executed App.clearInterval");'
    +'    reportDoc.info.title = "Bookmark Listings";'
    +'    reportDoc.info.Author = "A. C. Robot";'
    +'} catch (e) {console.println("Waiting...: " + e);}'
    , 100); // check every 1/10 th of a second, you can adjust this

```

Solution Notes: You can edit the script and customize the look for the report document: Add bullets at different levels, or only report on the top level bookmarks.

You can use the “TouchUp Object Tool” to copy and paste the column of bookmarks to make a two or three column format.

Count Bookmarks

Problem: Count the number of bookmarks in a given PDF document.

Solution:

```

/* Count Bookmarks */
/* Recursively work through bookmark tree */
function CountBookmarks(bm, nLevel)
{
    if (nLevel != 0) counter++        // don't count the root
    if (bm.children != null)
        for (var i = 0; i < bm.children.length; i++)
            CountBookmarks(bm.children[i], nLevel + 1);
}
var counter = 0;
CountBookmarks(this.bookmarkRoot, 0);
console.show();
console.println("\nFile: " + this.path);
console.println("The number of bookmarks: " + counter);

```

Solution Notes: This sequence is general enough that it can be run on any of the "Run commands on" options: "Files Open in Acrobat", "Selected Files", "Selected Folder", or "Ask When Sequence is Run".

Gather Bookmarks to Doc

Problem: Given a selection of PDF files (see [Selected Files](#)), *not open in the viewer*, create a series of bookmarks in the document open in the viewer that point to the selected files.

Solution:

```
/* Gather Bookmarks to a Doc */
/* Assumption: One document is open in the viewer. It is into this document
   bookmarks to the selected files are inserted.
*/
try
{
    // Get active docs in viewer.
    var d = app.activeDocs;
    if (d.length != 1) // if not exactly one doc, exit batch
        throw "One and only one file is required to be open in the Viewer";
    else // if there is a doc open, assume it is the one the user wants
        var tDoc = d[0];

    // Get the path of the file Acrobat has taken from the selected folder
    var path2file = this.path;

    // regular expression acquire the base name of file (1)
    var re = /\.*\|\/\|\.pdf$/ig;

    // filename is the base name of the file Acrobat is working on (2)
    var filename = path2file.replace(re, "");

    // If title is available, use it for a bookmark name, else use filename
    if ((this.info.Title == null) || (this.info.Title.replace(/\/s/g, "") == ""))
        var bmToFile = filename;
    else
        var bmToFile = this.info.Title;

    var nIndex = (this.bookmarkRoot.children == null )
        ? 0 : this.bookmarkRoot.children.length;

    // make a bookmark with action
    tDoc.bookmarkRoot.createChild(bmToFile,
        'if (this.external)\r\t'
        + 'this.getURL("'" + filename + '.pdf");\r'
        + 'else {\r\t'
            + 'var that = app.openDoc("'" + filename + '.pdf", this);\r\t'
            + 'if ( this != that ) this.closeDoc();\r'
        + '}', nIndex);
}
catch (e)
```

```

{
    console.println("Aborting Batch: " + e)
    event.rc = false;
}

```

Solution Notes: **Important:** To see the bookmarks, you may have to do a “Save As”.

- This sequence can be used by itself to insert a series of navigation bookmarks, or in conjunction with [Copy Bookmarks from Doc](#).
- This script strips away the path to the file, so the links will work if the PDF file containing bookmarks are in the same directory as the targeted files. This is suitable for use with the [Copy Bookmarks from Doc](#) sequence.
- If you want to reference files in different directories, then there must be a modification of the above script. Lines (1) and (2) would have to be reworked to reflect a more complex relative path. This is left as an exercise.

Copy Bookmarks from Array

Problem: Take the collection of bookmarks, gathered by [Gather Bookmarks to Array](#), and insert them into each of the [Selected Files](#).

Solution:

```

/* Copy Bookmarks from an Array */
/*
    Run "Gather bookmarks into an Array" first
    Uses: global.bmArray, global.bmFileCnt from the sequence
*/
// Begin Job. Executed when first file is processed.
if ( typeof global.counter == "undefined" ) {
    console.println("Begin Job...");
    global.counter = 0;
    // if Count PDF Files has not been run, then...
    if ( typeof global.FileCnt == "undefined" )
        // then use the file count from Gather bookmarks from an Array
        global.FileCnt = global.bmFileCnt;
    /* Note: A routine to sort the global.bmArray can be inserted here */
}

// This script is executed for each of the selected files
try
{
    global.counter++
    console.println("Processing file \#" + global.counter);

    // Now, insert a bookmark at end of bookmark tree
    for (var i = 0; i < global.bmArray.length; i++) {
        var nIndex = (this.bookmarkRoot.children == null ) ?

```

```

        0 : this.bookmarkRoot.children.length;
        this.bookmarkRoot.createChild(global.bmArray[i][0],
        'if (this.external) \r\t'
        +'this.getURL("'" + global.bmArray[i][1] + '.pdf");\r'
        +'else {\r\t'
        +'var that = app.openDoc("'" + global.bmArray[i][1]+'.pdf", this);\r\t'
        +'if ( this != that ) this.closeDoc();\r'
        +'}',
        nIndex);
    }
}
catch (e)
{
    console.println("Aborting Batch: " + e);
    console.println("Current File: " + this.path);
    event.rc = false;
    delete global.counter;
}

// End of Job. Executed when last file is processed.
if (global.counter == global.FileCnt) {
    console.println("End of Job...");
    delete global.indexCnt;
    delete global.bmFileCnt;
    delete global.bmArray;
    delete global.counter;
    delete global.FileCnt;
}

```

Copy Bookmarks from Doc

Problem: Take the bookmarks in the document in the viewer, and copy them to each of the [Selected Files](#).

Setup: Open a PDF document with a series of bookmarks in it you want to copy to the selected files in the Acrobat viewer. Perhaps the file containing the bookmarks was obtained by running the sequence [Gather Bookmarks to Doc](#). These bookmarks should have only “go to remote” actions associated with them. (Perhaps, the bookmarks are used for navigation between the selected of files.)

Before running this script select the files to which you want to transfer the set of bookmarks in the viewer. Your choice should be something other than “Files Open in Acrobat” as the one file open in the viewer contains the bookmarks to be transferred.

Important: In the batch sequence that follows, it is assumed that the document containing the bookmarks has *only one page*, and it is the only document open in the viewer.

Solution:

```
/* Copy Bookmarks from a Doc */
try
{
    var d = app.activeDocs;
    console.println("this.path = "+this.path);
    // we can have only one file open in the viewer, this is the master
    // bookmark list.
    if (d.length != 1)
        throw "Too many docs open, no bookmarks will be copied.";
    var tDoc = d[0];

    // insert the doc in viewer (tDoc), into the current doc (this)
    this.insertPages(-1,tDoc.path);

    // now delete the page just inserted
    this.deletePages(0);
}
catch(e)
{
    console.println("Aborting Batch: " + e);
    console.println("Current File: " + this.path);
    event.rc = false;
}
}
```

Solution Notes: The method of inserting and deleting pages to copy bookmarks is slower than the method of [Copy Bookmarks from Array](#). One advantage this method has over the other ([Copy Bookmarks from Array](#)) is the bookmarks can have arbitrary actions associated with them. (**Beware:** Some of these actions may not make sense when transferred to other files.)

Insert Bookmarked Pages

Problem: There is a number of one page PDF files (perhaps, image files) that needs to be merged into one file with bookmarks to each page.

Solution: Place a cover sheet, a PDF file with one or more pages, in the viewer. Make a selection of files, assumed to be one page. (They can be multiple pages, but only the first page of each file will be inserted.)

```
/* Insert Pages with Bookmarks */
/* Insert PDF page one from the selected files into the current
   document open in Acrobat. This open document is assumed to be the
   *only* document in the viewer.
*/
try
{
```

```

// Get active docs in viewer.
var d = app.activeDocs;
// The document into which we are going to insert pages is
// assumed to be d[0], since there is only one file open in the viewer.
if (d.length != 1) throw "Need exactly one file open in Viewer";
var tDoc = d[0];

// get the number of pages in the document
var lastpage = tDoc.numPages;

// Get the path of the file Acrobat has taken from the selected files
var path2file = this.path;

// regular expression to remove the path to the folder of this file
var re = /.*\|\.pdf$/ig;

// filename is the name of the file Acrobat is working on
var filename = path2file.replace(re, "");

console.println("Processing: " + filename);

// If title is available, use it for a bookmark, else use filename
if ((this.info.Title == null) || (this.info.Title.replace(/\s/g, "") == ""))
    var bmToFile = filename;
else
    var bmToFile = this.info.Title;

// Now, insert pages into the document open on the viewer.
// If you want the whole document inserted, then you would insert
// nEnd: this.numPages-1 in the argument list of the insertPages method
tDoc.insertPages
({
    nPage: lastpage - 1,
    cPath: path2file
});

// insert a bookmark
var nIndex = (tDoc.bookmarkRoot.children == null) ?
    0 : tDoc.bookmarkRoot.children.length;
tDoc.bookmarkRoot.createChild(bmToFile, "this.pageNum=" + lastpage, nIndex);
}
catch (e)
{
    console.println("Aborting Batch: " + e );
    console.println("Current File: " + this.path );
    event.rc = false;
}

```

```
}
```

Icons

Import Named Icons

Problem: Import a series of icons into the document currently in the Acrobat Viewer.

Solution: The assumptions of this solution are that there is only one document open in the Viewer; this is the document into which you want to import some named icons. The icons are the selected files. Each of the selected PDF files contains only one icon on the first page (page 0). It is assumed also that each icon file has a document title; this title is used to assign the icon a name. Here is the batch code.

```
/*Import Named Icons*/

/* This requires a file open in the viewer into which the icons
   will be inserted.
*/

try
{
    var d = app.activeDocs;
    if ( d.length != 1)
        throw "Batch requires a file to be open in viewer to insert Icons."
    var myDoc = d[0];           // target doc is only one open in Viewer

    // Get the path of the file Acrobat has taken from the selected folder
    var path2file = this.path;
    console.println("\nPath: " + path2file);

    // regular expression acquire the base name of file
    var re = /\.*\|\.pdf$/ig;

    // filename is the base name of the file Acrobat is working on
    var filename = path2file.replace(re,"");

    // If title is available, use it for the icon name, else use filename
    if ((this.info.Title==null) || (this.info.Title.replace(/\s/g,"")== ""))
        var nameIcon = filename;
    else
        var nameIcon = this.info.Title;

    console.println("Name used: " + nameIcon );

    // insert icons!
```

```

    myDoc.importIcon(nameIcon, this.path);
} catch (e) {
    console.println("Aborted Batch: " + e);
    event.rc = false; // abort batch
}

```

Solution Notes: Once the icons have been introduced into the document by name, it is easy to associate any of these icons with any particular button. For example,

```

var f, i;
f = this.getField("iconButton1");
i= this.getIcon("anIconName1");
f.buttonSetIcon(i);

f = this.getField("iconButton2");
i= this.getIcon("anIconName2");
f.buttonSetIcon(i);
... ..
... ..

```

These lines can be a part of another batch sequence which runs after the icons have been imported. Repetitive code such as above can then associate each button requiring one or more icons with one or more named icons.

You can remove the icons with the *Doc.removeIcon* method, then import new icons into the same document; this makes it fairly easy to maintain updates.

Insert Navigation Icons

Problem: Insert a set of navigational icons on each page of each file in the selected set of files.

Solution: The following script is a representative example of the type of code you need to write. You will have to modify it as appropriate.

```

/* Insert Navigation Icons */

// getPageBox, the default is "Crop"
var aPage = this.getPageBox();
var w = 36;           // width of each icon
var nNavi = 4;       // number of navigation icons to be placed
var g = 6;           // gap between icons
var totalWidth = nNavi * w + (nNavi - 1) * g; // total width of navi bar

var widthPage = aPage[2] - aPage[0];
// horizontal offset to left-most edge of navi bar needed to center it
var hoffset = (widthPage - totalWidth) / 2;
var voffset = 12;           // vertical offset from bottom

```

```

/* Import Icons into the document. This uses the Arrows.pdf distributed with
   the Acrobat 3.0 CD. But any set of icons will do. */
this.importIcon("PrevPage", "/F/Adobe/Batch/Arrows.pdf",48);
this.importIcon("PrevPage_p", "/F/Adobe/Batch/Arrows.pdf",52);
this.importIcon("NextPage", "/F/Adobe/Batch/Arrows.pdf",49);
this.importIcon("NextPage_p", "/F/Adobe/Batch/Arrows.pdf",53);
this.importIcon("PrevView", "/F/Adobe/Batch/Arrows.pdf",50);
this.importIcon("PrevView_p", "/F/Adobe/Batch/Arrows.pdf",54);
this.importIcon("NextView", "/F/Adobe/Batch/Arrows.pdf",51);
this.importIcon("NextView_p", "/F/Adobe/Batch/Arrows.pdf",55);

for ( var nPage = 0; nPage < this.numPages; nPage++) {
    // Create the fields
    var pp = this.addField("PrevPage", "button", nPage,
        [ hoffset, voffset, hoffset + w, voffset + w ] );
    var np = this.addField("NextPage", "button", nPage,
        [ hoffset + w + g, voffset, hoffset + 2*w + g, voffset + w ] );
    var pv = this.addField("PrevView", "button", nPage,
        [ hoffset + 2*w + 2*g, voffset, hoffset + 3*w + 2*g, voffset + w ] );
    var nv = this.addField("NextView", "button", nPage,
        [ hoffset + 3*w + 3*g, voffset, hoffset + 4*w + 3*g, voffset + w ] );

    // place fields objects in an array for convenience
    var NaviFields = new Array( pp, np, pv, nv );

    for ( var i = 0; i < NaviFields.length; i++ ) {

        var o = NaviFields[i];

        // Set the Properties
        o.buttonPosition = position.iconOnly;
        o.highlight = highlight.p;

        // Set appearance faces
        o.buttonSetIcon(this.getIcon(o.name),0);
        o.buttonSetIcon(this.getIcon(o.name+"_p"),1);
    }

    // Set Actions
    pp.setAction("MouseUp", "this.pageNum--");
    np.setAction("MouseUp", "this.pageNum++");
    pv.setAction("MouseUp", "app.goBack()");
    nv.setAction("MouseUp", "app.goForward()");
}

```

Extract Pages

Extract Pages to Folder

Problem: For each file from the [Selected Files](#), extract each of its pages and save the extracted pages to a particular folder.

Solution:

```
/* Extract Pages to Folder */
// regular expression acquire the base name of file
var re = /\.*\|\/\|.pdf$/ig;
var filename = this.path.replace(re,"");

try
{
    for ( var i = 0; i < this.numPages; i++ )
        this.extractPages
            ( {
                nStart: i,
                cPath: "/F/temp/"+filename+"_" + i + ".pdf" // (*)
            } );
}
catch (e)
{
    console.println("Batch Aborted: " + e )
}
```

Solution Notes: Before running this script, change line (*) appropriately. Change the path the folder; change the filename, and the method of referencing the i^{th} page of the target document.

Database

Two important features appear for the first time in Acrobat 5.0: ADBC and the new Batch Sequencing. These two can be combined in a variety of ways to perform a variety tasks previously reserved for CGI script.

Populate and Save

Problem: Given a SQL select criterion for a particular database, we want to take a PDF form, open in the viewer, and populate it with data from the database that satisfies the criterion, then save the form to a pre-selected folder.

The file *FormsDemo.pdf*, found in the *database* folder of the Acrobat 5.0, was the one used to develop a solution to this problem.

Instructions for use of *FormsDemo.pdf*:

- This demo uses the Microsoft Access database file *ADBCdemo.mdb*, found in the *database* folder of the Acrobat 5.0 CD. This Access file needs to be registered with the ODBC Data Source Administrator, see the instructions contained on the first page of the demo file *ADBCdemo.pdf*; or see the file *ADBC.pdf*, both of which are also contained in the *database* folder of the Acrobat CD.
- Move the batch sequence file “Populate and Save.sequ” to the users’ sequence folder. (If necessary, search of the word “Sequences” using *Start > Find > Files or Folders.*)
- After reading the solution below, open the file *FormsDemo.pdf* in the Acrobat viewer.
- Modify the text field named ”DIPath” to an appropriate folder on your own hard drive.
- Run the sequence “Populate and Save”. If all went well, copies of the forms, after they have been populated from the *ADBCdemo.mdb* should appear in the folder you specified.

Solution: The approach to the solution to this problem is to have a fairly general batch sequence and an “intelligent” form. The form contains, in addition to the fields pertinent to the use of the form, four additional fields that the batch routine uses:

- **DB :** This text field contains the name of the database to be used for retrieving data; for example: *ADBCdemo*.
- **SQL :** This is a text field that contains a SQL select command for choosing data from a database, as defined in the "DB" field; for example: *Select * from ClientData Where Income > 100000*.
- **DIPath :** This is a text field that contains a device independent path to the folder in which the saved PDF files are to be deposited; for example: */F/database/*. The path can contain a suffix; for example, */F/database/ID*. The DIPath will be concatenated to the NameRule, below, to obtain the complete path name of the file to be saved.
- **NameRule :** A unique filename to give the form about to be saved. Each row of data retrieved from the database should contain a field that uniquely characterizes the record. Perhaps, its a client ID. The ID can be used to build the name of the output file; for example, the *ADBCdemo* database has an ID field for each data row, so we can put the value of the text field "NameRule" to *row.ID.value*. When the batch sequence is run, each file will have a pathname of */F/database/ID1.pdf, /F/database/ID2.pdf, etc.*

The form also contains four document-level JavaScript functions which the batch sequence, listed below, calls.

- **getConnected():** This function reads the values of "DB" and "SQL", connects to the database and executes the requested SQL.
- **getNextRow():** This function simply gets the next row of data, it returns either a row object, or null.
- **populateForm(row):** This function populates the form with the row of data passed to it.
- **saveToFolder():** This function saves the populated form to the path obtained by concatenating the value of "DIPath" and the evaluated value of "NameRule".

Listed below is the JavaScript batch sequence named *Populate and Save*:

```

/* Populate and Save */
try
{
    // This counter is not really needed, but can be used to count the number
    // of files processed, or use it while in a testing stage.
    var counter=0;

```

```

        if ( !getConnected() ) throw "Could not connect!";
        do
        {
//          if (counter > 2) throw "Bailing out!"          // debug line for testing
          var row = getNextRow();
          if ( row == null ) throw "No more rows. Total processed: " + counter;
          counter++
          populateForm(row);
          saveToFolder();
        } while (true)
    }
    catch(e)
    {
        console.println("Batch aborted: " + e);
        event.rc = false;    // abort batch
    }

    console.println("End of Job Reached!");
    this.closeDoc(false);    // close the last form that was populated.

```

Security

Change Security to None

Problem: Given the [Selected Files](#) password protected files *all with the same password*, we want to remove the password protection.

Solution: The problem can be solved using the **Security** hard-wired sequence. If you edit the sequence, you'll see that it is set on "None" ("No Security").

To make this sequence run correctly, you must first go to the *Edit > Preferences > General > Batch Processing* dialog and change the security handler to "Acrobat Standard Security". When you run the sequence, Acrobat will ask for a password. Once that is entered, the batch will continue, change security to "None".

Gather DigSig Info

Problem: Write a report of the state of all digital signatures in the [Selected Files](#).

Solution: This sequence is rather long to be listed in this document. This is again an example of cross-document processing and requires that the *global.FileCnt* be set (usually by running [Count PDF Files](#)). The output is a report in which all signatures are grouped by their status: blank, invalid, unknown status and valid. A complete listing of digital signature information, using *Field.signatureInfo*, is given. If a signature is blank, invalid or has an unknown status, it is highlighted in red.

Signature Sign All

Problem: Given the [Selected Files](#), we want to create an invisible signature on each of the files.

Solution:

```
/* Signature Sign All */

// choose handler
var ppklite = security.getHandler("Adobe.PPKLite");

// login -- change as appropriate
ppklite.login("dps017", "/C/Profiles/DPSmith.apf");

// add signature field with zero dimensions -- invisible
var f = this.addField("Signature", "signature", 0, [0,0,0,0]);

// sign it and logout --- change as appropriate
f.signatureSign(ppklite,
  { password: "dps017",
    location: "San Jose, CA",
    reason: "I am approving this document",
    contactInfo: "dpsmith@mycompany.com",
    appearance: "DPSmith"});
ppklite.logout();
```

Solution Notes: Some people may want to set security—such as “No Changes”—and sign the document(s) in batch mode. *Security must be set before signing.* Edit the “Signature Sign All” batch (or a renamed copy of it): (1) Choose *File > Batch Processing > Edit Sequences...*; (2) from the list, choose “Signature Sign All” (or the sequence you have renamed it); (3) Click on the *Edit Sequence...* button, now click on the *Select Commands...* button; (4) Click on “Security” in the left-hand panel and move it over to the right-hand panel; (5) move “Security” up above the “Execute JavaScript” corresponding to “Signature Sign All”; (6) edit the “Security” batch sequence; (7) select “Acrobat Standard Security” then click on “Change Settings”; (8) choose the 128-bit RC4 encryption level and for example, “Only Form Fill-in or Signing” for changes allowed; (9) optionally, add in a password; (10) finally back out of all the dialogs by clicking on “Ok”, “Close”, and so on.

Insert Comments and Form Fields

Insert Stamp

Problem: Insert a stamp on the first page of each of the [Selected Files](#).

Solution: This script places the stamp in the geometric center of the page. The width of the stamp is set to 67% of the width of the cropped page; the height set at 25% of that.

Annots calculates all the coordinates in default user space. The form plug-in calculates the coordinates in rotated user space. Some undocumented JavaScript methods are used for that purpose.

Here is the batch script.

```
/* Insert Stamp */

// get crop box of first page
var aCrop = this.getPageBox()

var semiWidth  = aCrop[2]/2;
var semiHeight = aCrop[1]/2;

// center of the crop box.
var x =  semiWidth;
var y =  semiHeight;

// Make the width of the stamp roughly, .67% of cropped page width, and
// make height of stamp 25% of the stamp's width
semiWidth = 0.67*semiWidth;    // make width .67 of width of page
semiHeight = 0.25*semiWidth;  // make height .25  of semiWidth

var Rect = new Array (x-semiWidth,y-semiHeight,x+semiWidth,y+semiHeight);

// the matrix m is a matrix that transforms from rotated to default user space
var m = (new Matrix2D).fromRotated(this,0)

// and transform Rect from rotated to default user space
Rect = m.transform(Rect)

// Now add in our annot using the Rect array for the rect property
var annot = this.addAnnot
({
  page: 0,
  type: "Stamp",
  name: "myStamp",
  rect: Rect,
  contents: "This document has been approved.",
  AP: "Approved"
});
```

Insert Barcode

Problem: Insert a barcode on the first page of each of the [Selected Files](#).

Solution: The solution to this problem is incomplete. Details specific to your particular problem need to be inserted.

The problem of inserting a form at a specific location in a document is much easier than to insert an annotation (see the discussion in [Insert Stamp](#)). We operate in rotated user space only.

```
/* Insert Barcode */
/*
    This script inserts a text field on the first page of each
    the selected documents. The textFont is set to a barcode font;
    hence, a barcode will appear. Barcode font required.
*/
// getPageBox, the default is "Crop"
var aPage = this.getPageBox();

// put barcode in upper right-hand corner of page 0
// text field is 144 pts wide and 20 high
aPage[3] = aPage[1] - 20;
aPage[0] = aPage[2] - 144;

var f = this.addField("myText", "text", 0, aPage);
f.delay = true;
f.readonly = true;
f.alignment = "right";
f.textFont = "Code39Two";           // a barcode font is required here
f.fillColor = color.transparent;
// a fixed code is inserted, a real application, this would be obtained,
// perhaps, from a database, perhaps through the ADBC plug-in.
f.value = "ID-123-4567";
f.delay = false;
```

Spell Check

Spell Check a Document

Problem: It is desired to go through a PDF file (or selected PDF files) and mark each misspelled, or questionable word with a squiggly underline annotation. The contents of the annotation contain suggested alternative spellings.

Solution:

```
/* Spell Check a Document */
var ckWord,numWords, i, j;
for (var i = 0; i < this.numPages; i++ )
{
    numWords = this.getPageNumWords(i);
    for ( j = 0; j < numWords; j++)
    {
```

```

        ckWord = spell.checkWord(this.getPageNthWord(i,j))
        if ( ckWord != null )
        {
            annot = this.addAnnot
            ( {
                page: i,
                type: "Squiggly",
                quads:  this.getPageNthWordQuads(i,j),
                author: "A. C. Acrobat",
                contents: ckWord.toString()
            });
        }
    }
}

```

Solution Notes: You can use *Doc.spellDictionaryOrder* and *spell.dictionaryOrder* to determine the dictionaries used, and the order in which they are checked.

Comments

Cross Doc Comment Summary

Problem: Gather all comments contained in the [Selected Files](#), sorted by author, and create a combined report.

Solution: This script has both "Begin Job" and "End Job" sections and illustrates how to write and manage a more complex batch sequence.

The script uses the global variable `global.FileCnt`. You can either run [Count PDF Files](#), which sets this value, or you can enter the value through the console before you run the sequence.

Be sure `global.counter` is undefined before you start this sequence; type `delete global.counter` in the console to do so. The sequence does execute "delete `global.counter`" if the batch aborts or if the batch finishes successfully.

```

/* Cross Doc Comment Summary */
/* Requires "Count PDF Files" to be run first to give
   global.FileCnt a value (or enter through console)
*/
// The Begin Job Routine. Executed while first file is processed.
if ( typeof global.counter == "undefined" ) {
    console.println("Begin Job");
    console.clear();
    global.counter = 0;
    // insert beginJob code here
    global.rep = new Report();
    global.rep.size = 1.2;
}

```

```

    global.rep.style = "NoteTitle";
}

// This script is executed for each of the selected files
try{
    // insert regular batch code here.
    global.counter++
    console.println("global.counter = " + global.counter)
    global.absindent = 0;
    this.syncAnnotScan();
    global.annots=this.getAnnots({nSortBy: ANSB_Author});
    global.rep.color = color.blue;
    if (global.counter != 1) global.rep.writeText(" ");
    global.rep.writeText("Annotation Summary: By Author");
    global.rep.writeText("Comments from file: "+this.path);
    global.rep.color = color.black;
    global.rep.writeText(" ");
    console.println("annots.length = "+global.annots.length);
    global.rep.writeText("Number of Annots: "+global.annots.length);
    global.rep.writeText(" ");
    var msg = "\u00B7 (%s) page %s: \"%s\""; // \u00B7 unicode bullet
    var theAuthor=global.annots[0].author
    global.rep.writeText(theAuthor);
    global.rep.indent(20);
    for (var i=0; i < global.annots.length; i++)
    {
        if (theAuthor != global.annots[i].author) {
            theAuthor = global.annots[i].author;
            global.rep.writeText(" ");
            global.rep.outdent(20);
            global.rep.writeText(theAuthor);
            global.rep.indent(20);
        }
        global.rep.writeText(util.printf(msg,
            global.annots[i].type, 1+global.annots[i].page,
            global.annots[i].contents));
    }
    global.rep.outdent(20);
} catch(e) {
    console.println("Aborted on run number " + global.counter);
    delete global.counter;
    event.rc = false; // abort batch
}

// The End Job Routine. Executed while last file is being processed.
if (global.counter == global.FileCnt) {

```

```

console.println("End Job");
// insert endJob code here
console.println("Before open report");

/* We now want to open our report (Report.open), but we cannot open or
   save (Report.save) a report while a modal dialog box is open, and
   there is on while batch running. So, we'll wait until batch is done,
   then we'll open the report. This step could be a problem child, if not
   set up correctly.
*/
*/
global.wrtRep = app.setInterval(
    'try {'
    +'      docRep = global.rep.open("myreport.pdf");'
    +'      console.println("Executed Report.open");' // debug
    +'      app.clearInterval(global.wrtRep);'
    +'      delete global.wrtRep;'
    +'      console.println("Executed App.clearInterval");' // debug
    +'      docRep.info.Title = "End of the Month Report: August 2000";'
    +'      docRep.info.Subject = "Summary of comments: August meeting.'"
    +'      docRep.info.Author = "A. C. Robot";'
    +'} catch (e) { console.println("Waiting...: " + e);}'
    +'finally { delete global.counter;'
    +'      console.println("Executed delete global.counter"); }'
    , 100);
}

```

Solution Notes: The information written to the report can and should be customized. Use this sequence as the starting point for designing your own report summary of comments.

There is a hard-wired “Summarize Comments” batch sequence as well. This will summarize comments of each of the selected files; however, the comments are not combined: the comments from the file *myDoc.pdf*, for example, are saved in a separate file named *myDoc_sum.pdf*. All these summary documents can then be merged together, if desired.