Given the lack of freely available examples and straight forward procedures on implementing PDF Templates, I have compiled this document in hopes of consolidating enough useful information to help someone devise an effective procedure for implementing PDF Templates.

This document has been left unprotected so that anyone wishing to make improvements, additions, include notes or make any other productive contribution can do so.

Please be kind enough to forward me a copy of any revised version you may create, so I can keep up with the developments. Send it to: pammy@nbnet.nb.ca Thanks!

Although I am ACE certified (Acrobat 3 at the time of this writing), my programming skills are very out-dated and insufficient for writing any usable code. So, if there are any programmers who wish to include some good code samples (well documented please), they would be greatly appreciated.

My main goal is to try to untangle the maze of freely available information on PDF Forms, and to extract only what is pertinent to PDF Templates. Thereby creating an exhaustive reference on PDF Templates only. I believe this tangled maze of information coupled with so many possible scenarios has unnecessarily complicated things for those who wish to exploit the great power of PDF Templates.

I hope this document helps.

Bryan Guignard November 1998 Version 1.0

PDF TEMPLATES

Introduction

Pre-requisites
Readers are assumed to know the following:

The process of creating form fields and buttons, and modifying their properties.
To take full advantage of the capabilities of Acrobat Forms, a knowledge of FDF is required.

FDF is described in an appendix of the Portable Document Format Reference Manual.

The Acrobat Forms Author lets you define a page(s) in your PDF document as a template(s), which can then be used to generate, or spawn, new duplicate PDF pages on the fly. Templates provide powerful form capabilities in three main ways:

Templates allow the user to fill out as many form pages as needed, spawning additional pages (complete with new form fields) on- the- fly. See the Acrobat Forms JavaScript Objects Specification for information on defining an action that spawns new pages.

If you are generating a form by importing data from a database, you can spawn as many pages as needed to contain different quantities of data.

You can use template pages as button icons in another form by invoking the template names from an FDF file.

THE BASICS

After a user has filled in an Acrobat form, they must click on a button whose action is a submit form action in order to submit the data to a server. The format of the submitted data may be either HTML-compatible (urlencoded) or FDF. The selection of which format to use is made at the time the form is created, in the same dialog box in which the form's creator enters the URL to which the data is submitted. Templates is an Exchange-only feature: it does not work in the free Reader.

If HTML is selected, the format is identical to and compatible with an existing HTML form. Existing CGI scripts for HTML forms may be used to parse data in this format.

If FDF is selected, the data format is, not surprisingly, FDF.

The URL to submit to is not restricted to the http scheme. It can also be the mailto scheme, e.g. mailto:someuser@somecompany.com

In general, the Forms plug-ins are designed to be run in a Web environment. Every FDF file contains a reference to a PDF file that the data is intended for, designated with the **"/F"** key inside the FDF file (unless the FDF file is for the same Form that you submitted from). When Acrobat Exchange (with the Forms plug-in) is sent an FDF file, it opens the appropriate PDF file, and fills the form fields with the data from the FDF file. If the PDF file is referenced by a URL (for example, http://yourcompany.com/file.pdf), the FDF file must be sent in response to a submit action from a PDF form. This PDF form is displayed by Acrobat Exchange via a plug-in for a supported browser. FDF files containing URL references raise an "invalid file specification error " when opened in a stand-alone Acrobat Exchange (that is, when the PDF file is not being viewed inside a Web browser).

A user with Acrobat Exchange and the Acrobat Forms Author Plug-in can fill in and submit forms to a server. The Acrobat viewer must be running inside a browser to submit the data via the World Wide Web. It is also possible to submit the data to a "mailto:" address to put the FDF data in an e-mail. Once the user fills out the PDF-based form, the data can be submitted to the server for processing. Data can be submitted from a PDF form in either FDF (Forms Data Format) or HTML. Data can only be imported into a PDF form if it is in FDF format. The FDF format is a simple text file that has a structure based on the PDF file format.

With Acrobat 3.0, when importing an FDF that came back as a result of a **SubmitForm** action, if the Form currently being displayed is not the one specified in the **F** key of the FDF dictionary (which is an optional key), then that Form is first fetched, and then the FDF gets imported.

When exporting FDF, Acrobat 3.0 computes a relative path from the location the FDF is being stored, to the location the Form is in, and uses that as the value of the **F** key in the FDF dictionary.

A plug-in to Acrobat Exchange can programmatically import FDF data into a PDF file from a local file system using the HFT made available by the Forms plug-in. For more information, see the **"Acrobat Form Plug-In HFT Specification"** and the **"Acrobat Core API Overview".** There is currently no way to import FDF or create form fields using OLE Automation.

An issue that can be confusing for developers working with FDF is that an FDF file cannot open a PDF and dynamically add a page to it. An FDF file can either dynamically assemble a PDF document from Templates and fill in the form fields, or, open a PDF file and fill in the form fields, but cannot do both. **The difference is subtle, yet significant.** For this reason using the methods **FDFSetFile** and **FDFAddTemplate**, when constructing a single FDF file, will create an invalid FDF file.

To create an FDF file that dynamically creates a PDF file from Templates using the FDF Toolkit involves calling **FDFAddTemplate** for the first page followed by **FDFSetValue** (or any other **FDFSet** method) to specify the data to be filled into the form fields for that page. Calling **FDFAddTemplate** again creates a new page that can be setup like the previous page.

When passing field names to methods such as **FDFSetValue** or any other method taking a field name, use the original field name from the Template even if you passed true for **bRename** in **FDFAddTemplate**. Conceptually the fields are renamed after the data has populated the PDF Template and the field names will not match up if you use the renamed field name when creating the FDF.

TEMPLATE TIPS

1. At present, you must start from an Acrobat Form if the server returns a template-based FDF. This limitation will be removed.

2. An FDF can either cause an existing PDF to be opened using **FDFSetFile** or create a new PDF by spawning pages from templates contained in PDF documents specified by the FDF.
Do not use **FDFSetFile**, **FDFSetID**, or **FDFSetStatus**.

3. Structure your FDF-generating program like this:

```
FDFCreate
<begin loop>
    FDFAddTemplate
    FDFSetValue (or FDFSetFlags,
    FDFSetOpt, etc.)
<end loop>
FDFSaveToFile
FDFClose
```

4. **FDFAddTemplate**: Pass the URL of the PDF containing the template as parameter.

You can use URLs that are relative to the Acrobat Form from which the submit action took place. You may pass an empty string as the URL, in which case the template is expected to reside (possibly as a hidden template) inside the Acrobat Form from which the submit action took place.

5. **FDFSetValue** (or **FDFSetFlags**, **FDFSetOpt**, and so forth.) Remember to use the field names as they appear in the template and not the field names after the templates are spawned (as they will be renamed).

The purpose of renaming fields when spawning pages is to avoid conflicts when one template is spawned multiple times onto the same PDF, or when unrelated fields in various templates have the same name.
Fields get renamed to P< page number>.<template name>_<template number on this page>.<field name>. For example, P3.flowers_0.rose.

6. If you are choosing to rename fields and you are using Javascript, make sure you use the right field names. For example:

```
var cFldName = "Description";
var cMe = event. target. name;
if (cMe. substring( 0, 3) == "P1.") {
    cFldName = "P1.OrderForm_ 0." +
    cFldName ;
}
```

7. You can spawn multiple templates onto the same page (la overlays).

8. There is a bug in the current version of Forms that prevents calculations and formatting from initially taking place in the generated PDF. New values entered after that work fine. This will be fixed.

CONFIGURATION TIPS

1. If you are submitting from an Acrobat Form and the server returns FDF, then the URL must end in "# FDF". For example: <http://myserver/cgi-bin/myscript# FDF>

2. The URL may be relative (to the URL of the Form that you are submitting from).

3. You may set the SubmitForm URL to a **mailto:** scheme.

4. If your script is producing FDF, then you need to ensure that it emits the correct MIME type, that is, **application/vnd.fdf**.

5. If your server returns a static FDF file, as opposed to one dynamically generated by a script, then you may have to define **application/vnd.fdf** as a new MIME type on your server.

6. The value of the **"/F"** key may be relative (to the URL of the Form that you submitted from).

7. The returned FDF must include an **"/F"** key giving the absolute URL of the PDF that it is for.

8. The PDF will automatically get loaded by Acrobat upon opening the FDF.

FdfTk TIPS

SetValue

For radio buttons and checkboxes, pass true for the last parameter.
The newValue parameter must be either Off (to uncheck the checkbox), or a value that was entered as the "Export Value" when defining the properties of the field in Acrobat (to check the checkbox or radio button).

SetStatus

You can use this call to cause an alert to pop up at the client.
You can use a text field to display a message.
You can create a "status" field in your form that has no border and no background color, and is read-only, and it will be invisible until some text is written to it via FDF.

SetFile

Use this API to have the FDF point to the PDF that it is for.
Use it only if the Acrobat Form from which the submit action took place is not the one that the FDF is for.
This call is mutually exclusive with AddTemplate.
You can use a URL that is relative to the Acrobat Form from which the submit action took place.
If the submit occurred from an HTML form, you must use an absolute URL.

SetOpt

If you use this API (to change the options for a listbox or combobox), make sure you also call SetValue.

SetFlags
Here are some useful idioms:

Hide the field SetFlags(< field name>, FDFSetF, 2)
Show the field SetFlags(< field name>, FDFClrF, 2)
Make the field read-only SetFlags(< field name>, FDFSetFf, 1)
Make the field writable SetFlags(< field name>, FDFClearFf, 1)

FDFSetAP
Acrobat will only import an **/AP** from FDF into fields of type Button.
Only the **FDFNormalAP** will be imported (**FDFDownAP** and **FDFRolloverAP** will be ignored) unless the button that the **/AP** is being imported into has a "Highlight" of type Push.
Be aware that the new imported **/AP** will not show if the "Layout" for the button is of type "Text" only.
If the picture looks too small inside the button field with too much white space around it, once the FDF containing the new **/AP** is imported into the Acrobat Form, you may want to crop (using Acrobat Exchange) the PDF page used as the source of the **/AP**.

Set< some action> Action
Reprogram the form on the fly. Use these calls to dynamically change the action associated with a button. For example, **SetSubmitFormAction**, **SetResetFormAction**, and **SetJavaScriptAction**.

JAVA SCRIPT

Doc Object Property
numTemplates
Type: Integer Access: R
This property returns the number of templates in the document.

Doc Object Method
getNthTemplate
Parameters: nWhich
Returns: cName
Use this function to retrieve the name of a template within in the document.

Doc Object Method
spawnPageFromTemplate
Parameters: cTemplate, [iPage], [bRename]
Returns: nothing
Use this function with a template name, such as the ones returned by **getNthTemplate**. The optional parameter **iPage**, represents the page number (zero-based) into which the template will be spawned. If that page already exists, then the template contents are appended to that page. If **ipage** is omitted, a new page is created at the end of the document. The optional parameter **bRename**, is boolean that indicates whether fields should be renamed. The default for **bRename** is **true**.

Example:
```
var n = this.numTemplates;
var cTempl;
for (i = 0; i < n; i++) {
    cTempl = this.getNthTemplate(i);
    this.spawnPageFromTemplate(cTempl);
}
```

FORMS DATA FORMAT

FDF uses the MIME type **application/vnd.fdf**. On Windows and Unix it uses the ***.fdf** extension, and on the Mac it has the **'FDF '** file type.

The FDF Catalog Object

The value of the FDF key in the Catalog object is a dictionary.

FDF attributes

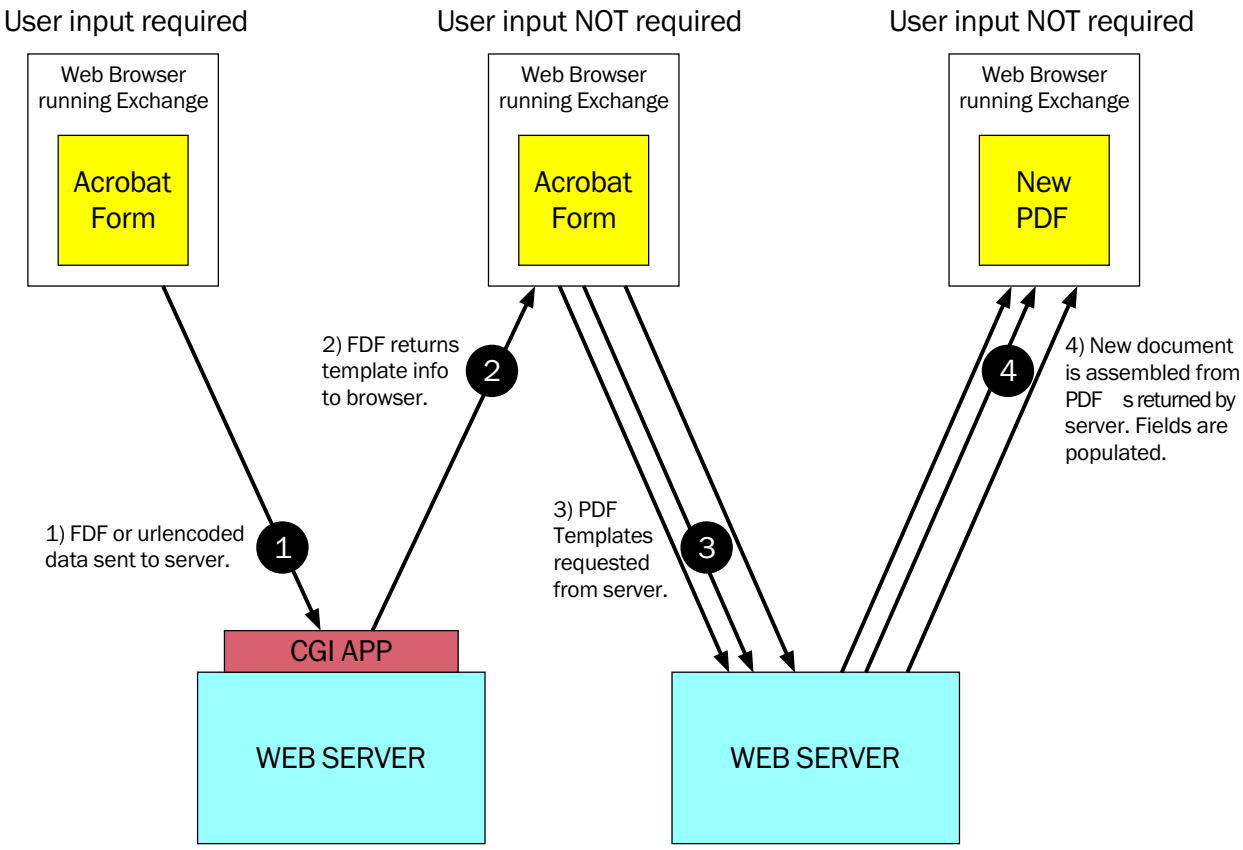
Key	Type	Semantics
Fields	array	(Optional) This array contains the root fields being exported or imported. A root field is one with no parent (i.e. it is not in the **Kids** array of another field).
Status	string	(Optional) A status to be displayed indicating the result of an action, typically a SubmitForm action. This string is encoded with **PDFDocEncoding**. The Acrobat 3.0 implementation of Forms displays the **Status**, if any, in an Alert Note, when importing an FDF.
F	file specification	(Optional) File specification for the Acrobat Form that this FDF was exported from, or is meant to be imported into.
ID	array	(Optional) The value of the **ID** field in the trailer dictionary of the Acrobat Form that this FDF was exported from, or is meant to be imported into.

FIELD ATTRIBUTES

Key	Type	Semantics
T	string	(Required) The partial field name.
Kids	array	(Optional) Contains the child field dictionaries.
V	various	(Optional) Field value.
Opt	array	(Optional) Options.
Ff	integer	(Optional) Field flags. When imported into an Acrobat Form, it replaces the current value of the **Ff** key in the corresponding field inside the Form. If **SetFf** and/or **ClrFf** are also present, they are ignored.
SetFf	integer	(Optional) Field flags. When imported into an Acrobat Form, it is **OR** ed with the current value of the **Ff** key in the corresponding field inside the Form.
ClrFf	integer	(Optional) Field flags. When imported into an Acrobat Form, for each bit that is set to one in this value, sets the corresponding bit in the Form field's **Ff** flags to zero. If **SetFf** is also present, **ClrFf** is applied after **SetFf**.
F	integer	(Optional) Widget annotation flags. When imported into an Acrobat Form, it replaces the current value of the **F** key in the corresponding field inside the Form. If **SetF** and/or **ClrF** are also present, they are ignored.

SetF	integer	(Optional) Widget annotation flags. When imported into an Acrobat Form, it is **OR** ed with the current value of the **F** key in the corresponding field inside the Form.
ClrF	integer	(Optional) Widget annotation flags. When imported into an Acrobat Form, for each bit that is set to one in this value, sets the corresponding bit in the Form's **F** flags to zero. If **SetF** is also present, **ClrF** is applied after **SetF**.
AP	dictionary	(Optional) Appearance of the Widget annotation.
AS	name	(Optional) Appearance state.
A	dictionary	(Optional) Action to be performed on activation of this Widget annotation.
AA	dictionary	(Optional) Additional actions.

PDF TEMPLATE GENERATION CYCLE



ACROBAT FORMS FIELD NAMING HINTS

What's in a Name?

A lot especially if you're authoring a PDF-based form. Careful name selection for fields can make form authoring and data collection easy, poor name selection can make it a downright chore. After all, it's tough enough filling out forms let alone authoring one. Enough said, let's talk about field names and how best to use them. We'll cover naming and how it changes the way that form fields are organized into a hierarchy within the document, how to take advantage of this hierarchy in JavaScript, and the Personal Field Name Specification and how it can make a form filler's life easier.

Data Sharing

A useful property about Acrobat Form fields is that fields that share the same name also share the same value. They can have different presentations of that data; they can appear on different pages, be rotated differently, have a different font or background color, etc. but they have the same value. This means that if you modify one field all others with the same name get updated automatically. In some forms packages, you have to perform a lot of work to get this to happen correctly. In Acrobat Forms, this happens pretty much for free.

Form Field Hierarchies

Typically, form fields have names like FirstName, LastName, etc. They are what we call flat names, there is no association between these fields. For many form applications, this flat hierarchy of names is sufficient and works well. By tweaking the field names slightly, we can create a hierarchy of fields within the document. For example, if we change them to Name.First and Name.Last we form a tree of fields. The period (.) separator in Acrobat Forms is used to denote a hierarchy shift. The Name portion of these fields is the parent, and First and Last are the children.

There is no limit to the depth of a hierarchy that can be constructed but it is important that the hierarchy remain manageable. It is also important to clarify some terminology: the field Name is known as an internal field (that is, it has no visible manifestation) and the fields First and Last are terminal fields (and show up on the page).

Note that this hierarchy can be useful in a number of ways including speeding up authoring and easier manipulation of groups of fields via JavaScript. In addition, a form field hierarchy can improve the performance of the forms engine if there are many fields in the form.

Name Clashes

One of the questions frequently asked is why you get the cryptic "You have chosen a name for your field that conflicts with an existing field". Either choose a new name or change the type of your field to match the existing field alert dialog. As shown in the above diagram, you can't have two terminal fields where one is named Name.First and the other is Name. The diagram illustrates that Name already exists as an internal field and so you can't have it as a terminal field as well (i.e. one drawn on the screen). You will also get this message if you have two fields with the same name and try to change one of them to a different type of field (i.e. text, radio, check, combo, button, list box). Fields that have the same name must also be of the same type.

Authoring

Acrobat Forms has the ability to automatically increment or decrement field names if they conform to a certain style. This feature helps to quickly copy and rename fields in a table. It is typical for cells in a table to be named with a numeric suffix (e.g. .0, .1, etc.). Acrobat Forms recognizes this type of numeric suffix and allows you to rename fields using the plus (+) and minus (-) keys when the field is selected without having to go to the properties dialog.

JavaScript

One of the other uses of field hierarchies is the ability to manipulate a group of fields using a single JavaScript statement. Given our original flat names for FirstName and LastName, imagine that we want to change the background color of these fields to yellow (to indicate missing data, perhaps). We would need two lines of JavaScript code per field.

```
var name = this.getField(FirstName);
name.bgColor = color.red;
var name = this.getField(LastName);
name.bgColor = color.red;
```

Given our hierarchy of Name.First and Name.Last above (and perhaps, Name.Middle), we can change the background color of these fields with one line of code:

```
var name = this.getField(Name);
name.bgColor = color.red;
```

This same technique can be used to make fields read-only or hidden or to change any of the other properties available for manipulation through JavaScript. This same naming scheme is also handy for summation and other tasks. The built-in scripts for Acrobat Forms can take advantage of your naming schemes to make summing an entire column of figures easy. In our invoice example above, you can calculate the value of a field GrandTotal by summing the totals using the old method and a custom calculation script:

```
var total = 0.0;
for (i = 0; i < 2; i++) {
    var f = this.getField(Total. + i);
    total += f.value;
}
event.value = total;
```

Alternatively, you can use the simple calculation feature and author the GrandTotal using just the user interface; no custom script is necessary.

Personal Field Names

How many times have you filled out your social security number on a form? Too many, I'm sure. So Adobe has come up with a way of naming fields, called the Personal Field Names Specification, that can make it such that you never have to type your social security number and other common data ever again. There is a document (PFNForm.pdf) that comes on your Acrobat Forms Maker CD in the Samples/Adobe/PFN folder that allows a user to enter in all the data they've ever had to enter (just one last time) and save this data to their local hard drive. Any form that they come across that conforms to the Personal Field Names (PFN) Specification allows them to instantly populate that form with their saved data. Sounds simple, right?

Well, as a forms author, you have to be careful about how you name fields (yes, the whole naming thing again). The PFN document lists several common fields and their suggested names. Corporations might want to add their own names to this list (e.g. Adobe.CostCenter) using a prefix that is corporate-specific. As an author, you also have to add a special button to the document that has the PFN logo and an action to import a user's profile from disk. A little extra work for a lot of convenience for your form user. A standard name for fields also makes it easier on the back-end process to collect data from a variety of forms and process it in similar ways. Your webmaster will be much happier treating form data in a consistent manner.

CATALOG (Adobe sample) SOURCE CODE

```
/*
* Catalog.c
*
* Copyright (C) 1997 by Adobe Systems Incorporated.
* All rights reserved.
*/

/*
  For more information about FDF, check out APPENDIX H, titled
  "Forms Data Format", of the "PDF 1.2 Reference Manual"
  http://www.adobe.com/supportservice/devrelations/PDFS/TN/PDFSPEC.PDF

  To report bugs in the FDF toolkit, e-mail acrodevsup@adobe.com
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef SOLARIS
#include <thread.h>
#endif
```

```
#include "FdfTk.h"

/*
  These definitions are needed only on Unix, for multithreading
*/
#ifdef SOLARIS
#define INITIALIZE_FDF_THREADSAFE { \
   static mutex_t mutexObj; \
   mutex_init( &mutexObj, NULL, NULL ); \
   FDFRegisterThreadsafeCallbacks( \
      (ThreadsafeCallback)mutex_lock, \
      (ThreadsafeCallback)mutex_unlock, \
      (ThreadsafeCallback)mutex_destroy, &mutexObj ); }
#endif

#ifndef INITIALIZE_FDF_THREADSAFE
#define INITIALIZE_FDF_THREADSAFE
#endif



/*
  This sample application starts with the user viewing CatEnglish.pdf or
  CatFrench.pdf. There are two checkboxes that allow the user to select a
  category of products they are interested in viewing. There is also a pair
  of radio buttons that can be used to select the country (US or Canada),
```

```
  FDF is constructed at the server, and sent back to the client. The FDF gives
  (as the value of the /F key) the name of the AcroForm that needs to be fetched
  in order to import the FDF data into it. There are four possibilities: 2 or 4
  page catalog, and French or English.
*/


#define NUM_PRODUCTS 7

/*
  Each of these files contains the image of one item in the catalog. FDF carries
  each image from the server to the AcroForm at the client, as the new
  "appearance" (i.e. the value of the /AP key) of a (non-interactive) button
  field in the AcroForm.
  In my installation, the following files reside in the cgi-bin directory,
  together with the "catalog" executable itself, and
  have read permission for the user id that the web server is running as.
*/
static char const * ProdPicFileName [NUM_PRODUCTS] = {
    "clip.pdf",
    "clipboard.pdf",
    "folder.pdf",
    "folderexpand.pdf",
    "folderopen.pdf",
    "stapler.pdf",
    "tapedispenser.pdf"
};
```

```
/*
  Product names (in English or French), to be used as the value of fields
  called "name1", "name2", etc.
*/
static char const *ProdName [][NUM_PRODUCTS] = {
    {
     "BINDER CLIPS",
     "CLIPBOARD",
     "FILE FOLDER",
     "EXPANDING FILES",
     "OPEN TOP FILES",
     "STAPLER",
     "TAPE DISPENSER"
    },
    {
     "ATTACHES POUR CARTABLE",
     "PRESSE-PAPIER",
     "DOSSIER",
     "DOSSIER EXTENSIBLES",
     "OUVRIR PREMIERS FICHIERS",
     "BROCHEUSE",
     "ALIMENTATEUR DE RUBAN"
    }
};

/*
```

```
  Product descriptions (in English or French), to be used as the value of fields
  called "desc1", "desc2", etc.
*/
static char const *ProdDesc [][NUM_PRODUCTS] = {
    {
     "Handles can be hung, folded flat against clipped material, or removed for permanent
binding.",
     "Hardboard.",
     "11pt. long grained fibers for strength.",
     "With flap and elastic cord. 7/8 inch capacity pockets. Reinforced double-thick front and
black.",
     "25 pockets with two indexing systems",
     "Holds 200 standard staples",
     "No moving parts"
    },
    {
     "Poignee accrochables pliable et amovible si desire.",
     "Tableau.",
     "11pt. fibres rigides.",
     "Avec languette et elastique. Capacite de 3cm par pochette. Double epaisseur avant et
arriere.",
     "25 sachets avec deux systemes d'indexation",
     "Contient 200 broches",
     "Sans piece amovible"
    }
};
```

```
/*
  Product numbers, to be used as the value of fields called "num1", "num2", etc.
*/
static char const *ProdNum [NUM_PRODUCTS] = {
    "BC-1020",
    "HC-10A",
    "WJ-100-33",
    "WJ-102-33",
    "WJ-101-33",
    "SS-2001",
    "TD-347"
};

/*
  Product prices (in US dollars), to be used as the value of fields called
  "price1", "price2", etc. These need to get multiplied by a currency factor
  to convert to Canadian dollars.
*/
static const double ProdPrice [NUM_PRODUCTS] = {
    3.0,
    1.9,
    12.5,
    10.59,
    11.77,
    10.0,
    20.0
};
```

```
/*
  Some items are categorized as "desktop" and some as "folders", and the user
  selects on the initial AcroForm which category he wants to view.
*/
static const ASBool IsDesktop [NUM_PRODUCTS] = {
    true,
    true,
    false,
    false,
    false,
    true,
    true
};

/*
  Error message if the user doesn't select a category. This is carried by the
  FDF as the value of the /Status key, which causes an alert to pop-up at the
  client. This is not the only way to convey status. We could also use a text
  field for this purpose, possibly one that is normally hidden and becomes
  visible only if needed. Visibility of a field can be controlled via flags
  changed through FDF, e.g. FDFSetFlags(theFDF, "status field", FDFClrF, 2).
*/
static char const *SelectCategoryPlease [] = {
    "Please select \"Desktop Supplies\" and/or \"Folders\"",
    "Fourniture de bureau ou dossiers?"
};
```

```
/*
  The error handling strategy we use in this example is to return to the client
  an indication of which call into the FDF API failed, and what the error code
  was. This is probably a good approach during the debugging phase, but not for
  production systems!
*/
static void HandleError(FDFErc theErc, const char* funcName, const char* parameter,
                FDFDoc inFDF, FDFDoc outFDF)
{
    if (theErc != FDFErcOK) {

     if (inFDF) FDFClose(inFDF);
     if (outFDF) FDFClose(outFDF);
     printf("Content-type: text/plain\n\n");
       printf("Error %d in call to %s. Parameter: %s\n", theErc, funcName,
           parameter);

#ifdef UNIX_ENV /* Do not need to do this on Win32 */
     FDFFinalize();
#endif

#ifdef MAC_PLATFORM
   FDFFinalize();
#endif
```

```
      exit(1);
    }
}


void main(int argc, char *argv[], char *envp[])
{
    FDFDoc inFDF = NULL;
    FDFDoc outFDF = NULL;
    FDFErc theErc;
    char cBuf[255];
    char cCountry[5];
    int iCountry = 0;
    double fCurrencyFactor = 1.0;
    ASBool bWantsDesktop = false;
    ASBool bWantsFolder = false;
    ASInt32 howMany = atoi(getenv("CONTENT_LENGTH"));
    int i;
    int nItems = 0;

#ifdef UNIX_ENV
    /*
      On Unix only, need to define the platform-dependent functions used for
      threading.
      This particular example does not use threading, but we still wanted
      to show how the FDFRegisterThreadsafeCallbacks() API is used.
    */
```

```
    INITIALIZE_FDF_THREADSAFE;
    /*
      This example assumes a cgi-bin environment, where each invocation of
      the program spawns a new process. In a multi-threaded environment,
      FDFInitialize() should only be called once, by the first thread in the
      application. Once again, this is Unix only. On Win32 the initialization
      and multithreading stuff is taken care of automatically by the FdfTk.dll
    */
    FDFInitialize();
#endif

#ifdef MAC_PLATFORM
    FDFInitialize();
#endif

    /*
      Open the FDF data submitted from the client. This example assumes cgi-bin,
      where the POSTed data is available at stdin, and its length is in
      environment variable CONTENT_LENGTH.
    */
    theErc = FDFOpen("-", howMany, &inFDF);
    HandleError(theErc, "FDFOpen", getenv("CONTENT_LENGTH"), inFDF, outFDF);

    /*
      Get the value of the radio button called "country". The AcroForm was
      defined so that one button exports the value "ca", and the other "us".
    */
```

```
theErc = FDFGetValue(inFDF, "country", cCountry, sizeof(cCountry), &howMany);
HandleError(theErc, "FDFGetValue", "country", inFDF, outFDF);

if (strcmp(cCountry, "ca") == 0) {
 iCountry = 1;
 fCurrencyFactor = 1.354;
}

/*
  Get the value of the checkbox called "desktop". If it is checked, we get
  "on" as the value, else we get "off". We could've changed the export value
  of the "on" state to something else, but decided to use the default.
*/
theErc = FDFGetValue(inFDF, "desktop", cBuf, sizeof(cBuf), &howMany);
HandleError(theErc, "FDFGetValue", "desktop", inFDF, outFDF);

if (strcmp(cBuf, "on") == 0)
 bWantsDesktop = true;

/*
  Get the value of the checkbox called "folders".
*/
theErc = FDFGetValue(inFDF, "folders", cBuf, sizeof(cBuf), &howMany);
HandleError(theErc, "FDFGetValue", "folders", inFDF, outFDF);

if (strcmp(cBuf, "on") == 0)
 bWantsFolder = true;
```

```
/* Create a new FDF to send the response to the client */
theErc = FDFCreate(&outFDF);
HandleError(theErc, "FDFCreate", "none", inFDF, outFDF);

if (!bWantsFolder && !bWantsDesktop) {
 /*
   Send an error message if the user doesn't select a category. This is carried
   by the FDF as the value of the /Status key, which causes an alert to pop-up
   at the client.
 */
 theErc = FDFSetStatus(outFDF, SelectCategoryPlease[iCountry]);
 HandleError(theErc, "FDFSetStatus", "none", inFDF, outFDF);
}
else {
 /*
   Find the products that the user is interested in
 */
 for (i = 0; i < NUM_PRODUCTS; i++) {
     if ((IsDesktop[i] && bWantsDesktop) || (!IsDesktop[i] && bWantsFolder)) {
       char cFldName[10];
       nItems++;

       /* Set the description of the item */
       sprintf(cFldName, "desc%d", nItems);
       theErc = FDFSetValue(outFDF, cFldName, ProdDesc[iCountry][i], false);
       HandleError(theErc, "FDFSetValue", cFldName, inFDF, outFDF);
```

```
            /* Its name */
            sprintf(cFldName, "name%d", nItems);
            theErc = FDFSetValue(outFDF, cFldName, ProdName[iCountry][i], false);
            HandleError(theErc, "FDFSetValue", cFldName, inFDF, outFDF);

            /* Its catalog number */
            sprintf(cFldName, "num%d", nItems);
            theErc = FDFSetValue(outFDF, cFldName, ProdNum[i], false);
            HandleError(theErc, "FDFSetValue", cFldName, inFDF, outFDF);

            /* Its price in the right currency */
            sprintf(cFldName, "price%d", nItems);
            sprintf(cBuf, "$%.2f", ProdPrice[i] * fCurrencyFactor);
            theErc = FDFSetValue(outFDF, cFldName, cBuf, false);
            HandleError(theErc, "FDFSetValue", cFldName, inFDF, outFDF);

            /* And finally its picture, obtained from page 1 of a PDF file */
            sprintf(cFldName, "pic%d", nItems);
            theErc = FDFSetAP(outFDF, cFldName, FDFNormalAP, NULL, ProdPicFileName[i],
1);
            HandleError(theErc, "FDFSetAP", ProdPicFileName[i], inFDF, outFDF);
          }
        }

        /*
          The FDF gives (as the value of the /F key) the name of the AcroForm that
```

```
    needs to be fetched in order to import the FDF data into it. There are
    four possibilities: 2 or 4 page catalog, and French or English.
    In my installation, the files cat2u2.pdf, cat2c2.pdf, cat2u4.pdf and
    cat2c4.pdf reside in the same directory as CatEnglish.pdf and CatFrench.pdf,
    and have read permission for the user id that the web server is running as.

  */
  sprintf(cBuf, "cat2%c%d.pdf", cCountry[0],
       (bWantsDesktop && bWantsFolder) ? 4 : 2);
  theErc = FDFSetFile(outFDF, cBuf);
  HandleError(theErc, "FDFSetFile", cBuf, inFDF, outFDF);
}

/*
  Don't forget to emit the correct HTTP header for the MIME type.
  By the way, remember when setting the URL of a SubmitForm action, to end
  it in #FDF. For example, the URL used in CatEnglish.pdf and CatFrench.pdf
  to submit to this cgi-bin program is:
  http:/ada/acrosdk/cgi-bin/catalog.cgi#FDF
*/
printf("Content-type: application/vnd.fdf\n\n");
fflush(stdout); /* Emit the HTTP header now! */

FDFSave(outFDF, "-"); /* Now send the FDF to stdout */

FDFClose(inFDF); /* Clean up after yourself! */
FDFClose(outFDF);
```

```
#ifdef UNIX_ENV
    /*
      This example assumes a cgi-bin environment, where each invocation of
      the program spawns a new process. In a multi-threaded environment,
      FDFFinalize() should only be called once, by the last thread in the
      application. Once again, this is Unix only. On Win32 the initialization
      and multithreading stuff is taken care of automatically by the FdfTk.dll
    */
    FDFFinalize();
#endif

#ifdef MAC_PLATFORM
    FDFFinalize();
#endif

    exit(0);
}
```