Chapter 8
Integrating PDF Technology

The Adobe Acrobat Portable Document Form (PDF) is proven technology that allows a Visual FoxPro developer to enhance the output generated by their custom applications. This chapter will show how you can integrate PDFs, extend the presentation of Visual FoxPro reports, and allow users to input data through PDF files into a Visual FoxPro application.

Generating Acrobat Portable Document Form (PDF) files has become commonplace and is as simple as printing output to a printer. If your customers are anything like our customers, they are asking for more and more integration of PDF output with custom applications. The Adobe Acrobat website has a quote on it that we think bests describe the Acrobat technology:

"Adobe® Portable Document Format (PDF) is the open de facto standard for electronic document distribution worldwide. Adobe PDF is a universal file format that preserves all the fonts, formatting, graphics, and color of any source document, regardless of the application and platform used to create it. Adobe PDF files are compact and can be shared, viewed, navigated, and printed exactly as intended by anyone with free Adobe Acrobat® Reader® software. You can convert any document to Adobe PDF using Adobe Acrobat 5.0 software."

Adobe PDF files can be published and distributed anywhere: in print, attached to email, posted on Internet sites, distributed on CD-ROM, viewed on a Palm or Pocket PC device, or even displayed in a Visual FoxPro application using an ActiveX control provided by Adobe. In a nutshell, any information that can be printed to a Windows printer can be generated into a PDF file. The PDF files are typically smaller than their source files, and can be downloaded a page at a time for fast display on the Web.

PDF files also provide an alternative way of sharing documents and application output over a broad range of hardware and software platforms without sacrificing any formatting that can be lost using HTML.

Which version of Acrobat do I need?

Acrobat comes in three flavors, Reader, Approval, and the full featured (known as plain old Acrobat). Adobe Acrobat was at version 5.0 when this book was written.

The reader is available free of charge and can be downloaded from Adobe's website. The generated PDF file can be viewed by anyone who has the Adobe Acrobat Reader. The Adobe Acrobat Reader displays the PDF file for viewing and has a number of features that include printing of the document, searching for text, and emailing the file to someone else. Users who just view the output generated by a custom Visual FoxPro application in PDF format can use this flavor of the product. Acrobat Forms can also be submitted to a web process using the Reader version of the product.

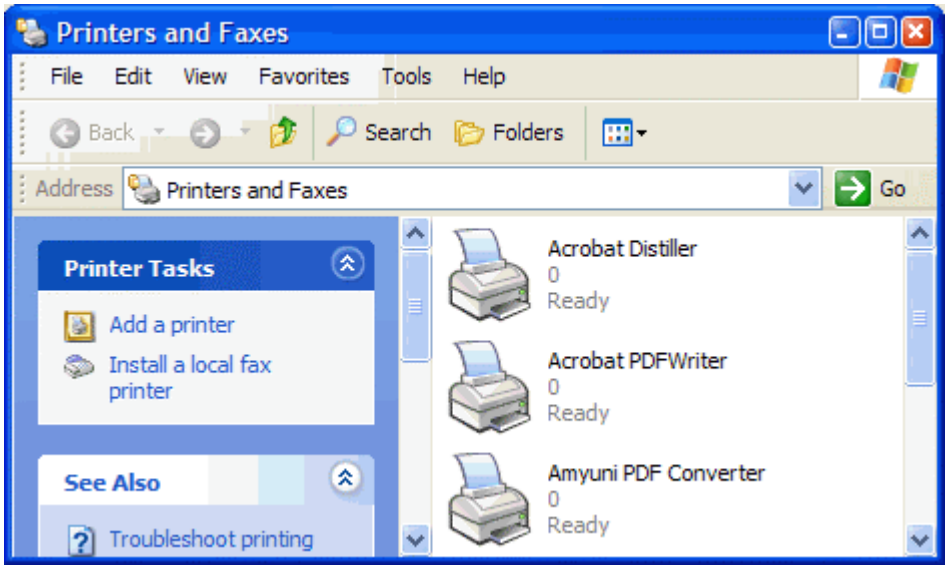
You need the full featured Acrobat application to be able to create PDF files, create Acrobat Forms, write JavaScript within a PDF, add electronic comments, or convert web pages to PDF. Custom applications developed with Visual FoxPro that create a PDF file using an Adobe product will need the full version of Acrobat.

##NOTE ICON
An individual Acrobat license is required for every workstation that will generate PDF files from your custom application. This means if you have 50 users working at 50 different workstations that access PDF generation functionality in the application, your customer will need 50 licenses at approximately US$225.

Acrobat Approval is available to save Acrobat Forms, apply e-signatures, spell check contents of a PDF, and to secure documents so others cannot make changes. If users are entering data into an Acrobat Form and need to save this data to the server or workstation hard drive, they can use this version of the product. Using Acrobat Approval can provide significant deployment savings if generating PDF files is not a feature that is required, but form data needs to be saved.

What is needed to generate a PDF file?

Acrobat PDF files are generated via a printer driver loaded on the client PC. These are printer drivers just like ones for a laser or color printer. These print drivers have the intelligence to generate files in the PDF format. As noted before, these files retain all the needed information to duplicate the output exactly as the original application intended it to be printed.



##IMAGE: MF08001.tif
Figure 8.1 *These are the printer drivers loaded when Acrobat and Amyuni drivers are installed.*

You can purchase the Acrobat product around US$225. When you install Acrobat (not the Reader) you get two printer drivers loaded. The PDFWriter is an older, less sophisticated driver. Distiller is the more powerful and more current driver. We have had good success with

the PDFWriter and find the limited features more than sufficient for our implementations. We have also found that it is faster in performance, which is good if the tradeoff of functionality is not limiting.

##NOTE ICON
If you plan to use the Acrobat PDF Writer driver you need to know that it is not loaded by default when installing Acrobat 5.0. You will need to select the custom setup and make sure to pick the PDF Writer to be installed.

One alternative to Acrobat that we have used successfully is the Amyuni PDF Converter (PDF Compatible Printer Driver). This runs $129 for a single-user license for one platform and $189 for all the Windows platforms (3.1, 95, 98, Me, NT, 2000, and XP). The Developer Version contains the ActiveX interface and is purchased one time ($800 for single OS platform, $1150 for all platforms) and has a royalty-free distribution license. The Developer Version only allows features to be accessed via the ActiveX interface and does not have any user interface, and no permanently loaded printer driver. This works well for Visual FoxPro (and other Visual Studio tools) based applications. The printer driver only exists at the time the driver is used and is generated on-the-fly when the ActiveX control is accessed to generate the PDF file. If your users need the user interface to the PDF Converter then they can get a site license for $2500 for a single OS platform or $3600 for all OS platforms. There is a new Professional version with encryption and web optimization available.

Okay, this sounds good so far, but wait there is more! Amyuni also has Visual FoxPro specific examples to boot and they actually advertise in Visual FoxPro periodicals! There is even more; they have even gone as far as developing an FLL API file for use with Visual FoxPro. Now the FLL solution is not always recommended since the ActiveX interface works well (unless you need bookmarks), but it is nice that Amyuni is showing support for Visual FoxPro in this fashion.

We are not trying to include an ad here for Amyuni, just trying to provide a baseline so you can evaluate the advantage or disadvantage of this product line. We advise you to check out the Amyuni.com website for all the details.

How do I determine which PDF product to license?

All PDF creation features are available in both the Adobe PDFWriter/Distiller and Amyuni PDF Converter drivers. The Amyuni PDF Converter gives an unlimited distribution product with the Developer Version. You or your client will need to purchase a full copy of Acrobat for every PC that will generate PDF files. In a small company (less than 6 users) it may be better to go the Acrobat route; larger sites or vertical market apps should seriously look at the Amyuni product. Adobe does have an Open Options Site License Program for organizations with 1,000 or more workstations. Contact Adobe for more specifics. Acrobat 5.0 also has the interactive development environment as well which may be something you or your customers will need.

Once the Acrobat printer driver is loaded it automatically becomes available to all Windows' applications and is actively visible in several applications already installed. For instance, all the Microsoft Office (v97, 2000, and XP) applications have the PDFMaker

macro/toolbar installed and available. The Amyuni version will not be available to other applications unless you get the site license.

There are other PDF writers available similar in functionality and implementation. We are most familiar with the Amyuni product which is why we have chosen it for discussion in this chapter. We are not endorsing this product over the others, just trying to express implementation ideas for these tools.

How can I use PDF technology in my Visual FoxPro apps?

An example of the use of these components is the company accountant publishing the sales results tracked in a custom database application (naturally developed by a top gun Visual FoxPro developer) to a PDF file. This file could be transferred via email to the sales force and they could view it on their laptops for review. Changes can be emailed back to accountant and updated in the database. The accountant recreates the PDF file and posts it on the company website. Now all employees in the company can hit the company website to see how well the company sales are going.

So why publish to the PDF format instead of HyperText Markup Language (HTML) format. HTML was designed for single page documents with limited formatting capabilities. The presentation of the document differs from one computer to another and from one web browser to another. Also, to transmit a single page, one needs to transmit many files containing different parts of the page (one file for each graphic). PDF documents can have hundreds of pages contained in one file with all the formatting capabilities that modern applications provide.

How do I output Visual FoxPro reports to PDF using Adobe Acrobat? *(Example: PromptPDF.prg)*

Once the full version of Adobe Acrobat is installed, generating Visual FoxPro reports to a PDF file is quite simple. First you make sure that the PDF Printer Driver is set as the default printer for the Visual FoxPro application. This can be any Visual FoxPro report. If the report has a hard-coded printer driver in the TAG, TAG2, and EXPR fields for a printer other than the Acrobat driver, the following code does not work. No special driver setting has to be made in advance, just use your standard methodology of outputting a report to the printer:

```
* Generic call where VFP prompts the user with the
* printer dialog each time the report is run
REPORT FORM ContactListing ;
       TO PRINTER PROMPT NOCONSOLE
```

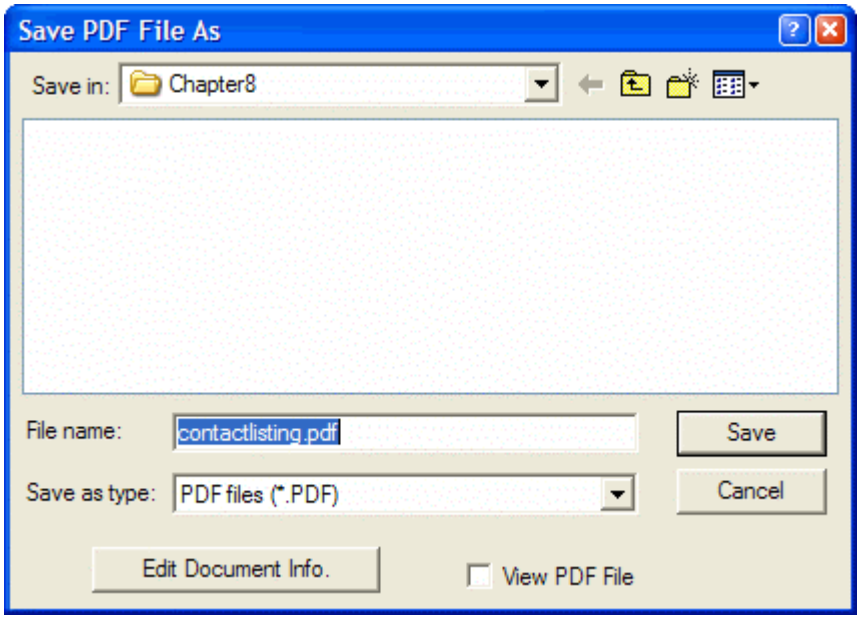
OR

```
* Generic call so user selects printer before
* report is printed, but it changes the VFP Printer
SYS(1037)
REPORT FORM ContactListing TO PRINTER NOCONSOLE
```

OR

```
* Call that has a hardcoded setting to drive the
* report to the Acrobat Printer, yet saves the
* old printer setting for reset later.
lcPDFPrinter = "Acrobat PDFWriter"
lcOldPrinter = SET("PRINTER", 2)

SET PRINTER TO NAME (lcPDFPrinter)

REPORT FORM ContactListing TO PRINTER NOCONSOLE

SET PRINTER TO NAME (lcOldPrinter)
```

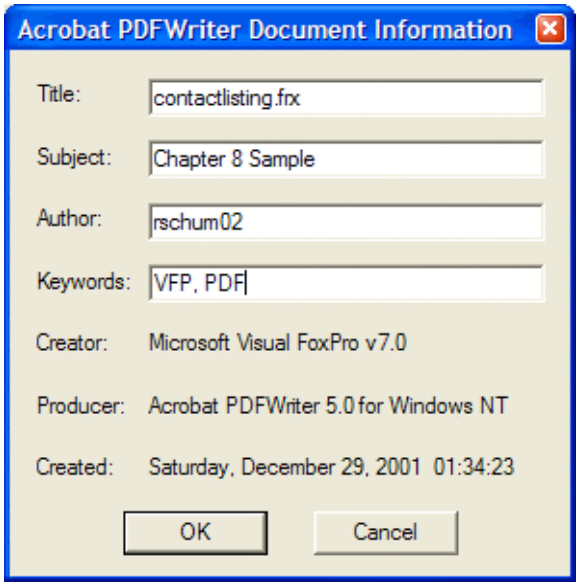
Once the report is sent to the printer via the REPORT FORM command, the following dialog is presented:



##IMAGE: MF08002.tif

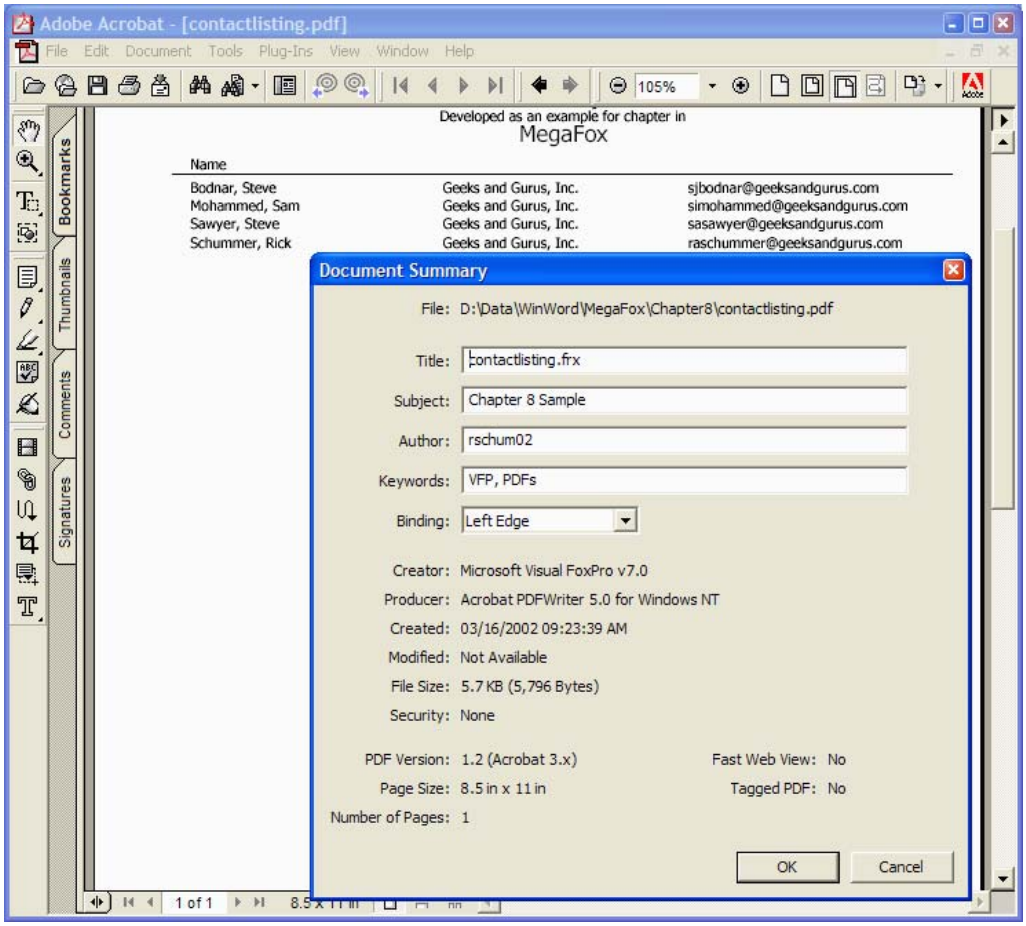
Figure 8.2 *The Save PDF File As dialog allows the user to specify the name of the PDF file as well as specific document properties.*

Optionally you can hit the Edit Document Info. commandbutton on this dialog to bring up the Acrobat PDFWriter Document Information dialog.



##IMAGE: MF08003.tif
Figure 8.3 *The PDF Document Information provides the readers of the document key details.*

This information is stored (and can be optionally reviewed) in the PDF file that is generated:



##IMAGE: MF08004.tif

Figure 8.4 *The Document Summary dialog within Acrobat will display the PDF Document information for the reader as entered by the document creator.*

The document summary information is often used by website search engines and indexers to make available the contents of PDF files to the people browsing their site.

What are the errors to trap when printing to PDFs? *(Example: cusAmyuniPDF::Error() of g2pdf.vcx, NoHandsAmyuniPdf.prg)*

The key to printing to PDFs (and any other printer driver selection process) is to capture the Visual FoxPro "Error loading printer driver" (error 1958). Make sure to include this trap in your error scheme or swap in a special error trap into the report printing mechanism.

```
LPARAMETERS tnError, tcMethod, tnLine

DO CASE
```

```
  CASE tnError = 1958
    THIS.lDriverError = .T.
  OTHERWISE
    AERROR(this.aErrorInfo)

    IF DODEFAULT(tnError, tcMethod, tnLine)
      MESSAGEBOX("There was a problem encountered when creating " + ;
                 "the PDF File (" + this.cPDFFileName + ")." + ;
                 CHR(13) + CHR(13) + ;
                 this.aErrorInfo[2] + " (" + ;
                 ALLTRIM(STR(this.aErrorInfo[1])) + ")", ;
                 0 + 48, _SCREEN.CAPTION)
    ENDIF
ENDCASE

RETURN
```

The biggest gotcha to watch when printing Visual FoxPro reports to PDF is getting bit by the hard coded printer details. One of the better-known problems with Visual FoxPro reports is accidentally hard coding printer driver information that gets stored in the report metadata. The information is stored in the report metadata file (FRX) in the EXPR, TAG, and TAG2 columns. If these fields have specific printer information included in the columns, Visual FoxPro will attempt to print to that printer and not the PDF driver. The symptom of this problem is having output printed on the printer when you attempt to generate a PDF file. We discussed this problem and a solution in 1001 Things You Wanted to Know About Visual FoxPro on page 542, “How to remove printer info in production reports”, and on page 496, “How to remove the printer information from Visual FoxPro reports”.

How do I run PDF reports unattended using Acrobat?

(Example: NoHandsPDF.prg)

In a previous section we discussed the basic Visual FoxPro report print to PDF process. While this process is straightforward, it has a significant drawback in the fact that it needs an end user to interact and enter in a file name before the PDF can be generated. What happens if you want to automatically generate a slew of reports from Visual FoxPro during a batch process that happens in the middle of the night? You or your clients could hire an operator that sits and watches the process and types in the filenames as they are prompted, or you can head directly to the West Winds web site and get the wwPDF50 ZIP file.

##WEB ICON

Rick Strahl has written plenty of code that allows Visual FoxPro developers to generate PDF files without the printer driver interaction prompting for a PDF filename. This class (wwPdf.prg) is available from http://www.west-wind.com/Webtools.asp and is available as part of the chapter source code downloadable from the Hentzenwerke website. The newest download available from West Wind has a change in it to better work with Acrobat 5.0. There are other classes included that work with Acrobat Distiller and the ActivePDF drivers.

The Acrobat printer driver is driven on settings available in the WIN.INI file. The wwPDF40 class manipulates the Acrobat filename settings in this INI file. The implementation of hands free Acrobat printing is straightforward:

```
* Partial listing from NoHandsPDF.prg
SET PROCEDURE TO wwPDF ADDITIVE
SET PROCEDURE TO wwAPI ADDITIVE

loPDF = CREATEOBJECT('wwPDF40')
lcFileName   = "ContactList" + lcNow + ".pdf"
lcOutputFile = ADDBS(SYS(2023)) + lcFileName

* Use PrintReport() instead of PrintReportToString()
* IMPORTANT: FRX must have printer specified as PDFWriter
loPDF.PrintReport("ContactListing", lcOutputFile)

* Destroy the PDF Object
loPDF = .NULL.
```

Set procedure to two programs that contain all the class definitions necessary to manipulate the needed operating system INI files that contain the information used by the Acrobat PDF printer driver. Then create the PDF file without the user being prompted for a file name. The sample code is creating the PDF output in the Visual FoxPro temp directory.

You might be wondering why the sample code has a **SET REPROCESS** command. The wwPDF classes work around an issue with the PDF Writer. The printer driver is single threaded. This means that it needs to generate one report at a time. The wwPDF class sets up a table and performs a record lock until the PDF is generated. This concept of enforcing the single threaded process is called semaphore locking. If you are simultaneously printing a massive amount of PDFs you might consider a different solution since this class will slow the overall throughput. Websites that generate PDF documents on the fly might want to consider the ActivePDF since it is multi-threaded and can take advantage of multiple processors.

There is a complete whitepaper on this topic written by Rick Strahl “Web reports with Adobe Acrobat Documents” at http://www.west-wind.com/presentations/pdfwriter/pdfwriter.htm. Rick Strahl also details how the PDF writing process is single threaded (important on a web server process) and exactly how his classes work with semaphore locking to make sure that the reports are handled one by one. If your web application is generating thousands upon thousands of Visual FoxPro reports in this manner the throughput may become an issue.

How do I run PDF reports unattended using Amyuni?

(Example: NoHandsAmyuniPDF.prg, cusAmyuniPDF::g2pdf.vcx)

In the previous section we demonstrated building PDF files in a hands-off mode (requiring no user interaction). This technique requires two tools, the full Acrobat version and the West Winds PDF classes. Rick Strahl is kind enough to offer his classes for free, but the Acrobat product lists for approximately $225 a license. If you are running this solution you need to buy a license for each user (or web server) that is generating these documents. This may not sound bad for a shrink-wrapped package that costs in the tens of thousands of dollars, but what if all 50 users need this functionality? You could be adding another $10,000 to the project

implementation costs. This is where a product like the Amyuni PDF Converter comes into play.

##NOTE ICON
Amyuni provides a full demonstration version of the Amyuni PDF Converter. We have included it in the chapter downloads, but a more current version might be available at the Amyuni website (www.Amyuni.com). The file name in the downloads is PdfSUDemoEn.exe. This needs to be installed to run the samples. The only difference between the demo and registered version is that a watermark is included on each PDF generated with the demo version.

The PDF Converter is accessed in code via an ActiveX interface or a FLL library. The examples we will demonstrate here is for the ActiveX interface. The class example (cusAmyuniPDF class in G2Pdf.vcx, available in the chapter download file from Hentzenwerke.com) handles both editions so feel free to review the code for the differences between the two approaches. First you must instantiate the control and initialize it.

```
this.oPDFPrinter = CREATEOBJECT("CDINTF.CDINTF")
this.oPDFPrinter.DriverInit("PDF Compatible Printer Driver")
```

After the printer driver is initialized we need to set up the parameters to achieve the desired output. This process is handled through the SetDriverParameter() method. There are several parameters available. We have set up several properties in the cusAmyuniPDF custom class to handle the options. The method code is as follows:

```
* cusAmyuniPDF.SetDriverParameter() method
* Do not prompt for file name
#DEFINE ccPDF_NOPROMPT            1
* Use file name set by SetDefaultFileName
* else use document name
#DEFINE ccPDF_USEFILENAME         2
* Concatenate files, do not overwrite
#DEFINE ccPDF_CONCATENATE         4
* Disable page content compression
#DEFINE ccPDF_DISABLECOMPRESSION  8
* Embed fonts used in the input document
#DEFINE ccPDF_EMBEDFONTS          16
* Enable broadcasting of PDF events
#DEFINE ccPDF_BROADCASTMESSAGES   32

IF NOT ISNULL(this.oPDFPrinter)
  * Set the destination file name.
  this.oPDFPrinter.DefaultFileName = this.cPDFFileName

  * Set resolution to to the desired quality
  this.oPDFPrinter.Resolution      = this.nResolution

  * Update driver info with resolution information
  this.oPDFPrinter.SetDefaultConfig()

  * Note: Message broadcasting should be enabled
  * in order to insert bookmarks from VFP.
  * But see the notes in the SetBookmark method
```

```
  this.oPDFPrinter.FileNameOptions = ;
    IIF(this.lPrompt, 0, ccPDF_NOPROMPT + ccPDF_USEFILENAME) + ;
    IIF(this.lBookmarks, ccPDF_BROADCASTMESSAGES, 0) + ;
    IIF(this.lConcatenate, ccPDF_CONCATENATE, 0) + ;
    IIF(this.lCompression, 0, ccPDF_DISABLECOMPRESSION) + ;
    IIF(this.lEmbedFonts, ccPDF_EMBEDFONT, 0)

  * Save the current Windows default printer
  * so we can restore it later.
  this.oPDFPrinter.SetDefaultPrinter()
ELSE
  * Handle settings via the FLL.
ENDIF

RETURN
```

Now the driver is ready to produce the PDF file. At this point you have made settings to have the user not prompted for a filename (default in this example), whether bookmarks are generated (FLL option only), if the contents are concatenated with previous output, if the PDF is compressed (a default for PDFs), and if fonts are embedded. This is not that much work. The Visual FoxPro report can now be generated with the following code:

```
* Set the VFP printer name to the PDF printer, and print the report.
this.cOldPrinterName = SET("printer", 2)
SET PRINTER TO NAME (THIS.cAmyuniDriver)
REPORT FORM (this.cReportName) NOEJECT NOCONSOLE TO PRINTER
```

The class also handles the resetting of the original printer driver and cleans up the object references in the Destroy method of the object. Modifications or enhancements to this class could also forward a text file or HTML output generated from your applications to a PDF file as well. Amyuni has other drivers available to support creation of HTML and text (via the Rich Text Format).

If you are using the Amyuni FLL interface you will need the FllIntf.fll file provided by Amyuni. This file is installed in the same directory as the Amyuni ActiveX controls and sample files. Even if you are not using the FLL interface you will need to include this directory in the Visual FoxPro path to recompile the class since the code is included for this option and the FLL is referenced.

How do I email a Visual FoxPro report? *(Example: MailPDFBatch.prg)*

One question that gets asked frequently on the support forums is: How can I email the results of a report? One approach is to run the Visual FoxPro report to a PDF file and have the application attach it to an email. There are a number of email components available that integrate with Visual FoxPro. It is beyond the scope of this chapter to get into the nuts and bolts of automating a MAPI compliant email client, but we wanted to reveal one of the most useful implementations of Acrobat PDFs in our applications. There are numerous examples of integrating email with Visual FoxPro in Chapter 4: Sending and Receiving E-mail. This example will leverage another class from West Winds called wwIPStuff.

Listing 8.1 A program that uses the wwIPStuff class and DLL from West Wind to email a Visual FoxPro report as a PDF file.

```
LPARAMETERS tlEmail

#INCLUDE foxpro.h

SET EXCLUSIVE OFF
SET DELETED ON
SET PROCEDURE TO wwPDF ADDITIVE
SET PROCEDURE TO wwAPI ADDITIVE
SET PROCEDURE TO wwUtils ADDITIVE
SET PROCEDURE TO wwEval ADDITIVE
SET CLASSLIB  TO wwIPStuff ADDITIVE

OPEN DATABASE pdfsample
SET DATABASE TO pdfsample

IF NOT USED("curMailing")
  USE pdfsample!v_geekscontactlist IN 0 AGAIN ALIAS curMailing
ELSE
  REQUERY("curMailing")
ENDIF

IF NOT USED("curList")
  USE pdfsample!v_geekscontactlist IN 0 AGAIN ALIAS curList
ELSE
  REQUERY("curMailing")
ENDIF

IF NOT USED("EmailInfo")
  USE pdfsample!EmailInfo IN 0 AGAIN ALIAS EmailInfo
ENDIF

IF NOT USED("EmailHistory")
  USE pdfsample!EmailHistory IN 0 AGAIN ALIAS EmailHistory
ENDIF

loIPMail = CREATEOBJECT('wwIPStuff')
loPDF    = CREATEOBJECT('wwPDF40')

SELECT curMailing

SCAN
  lcFileName   = ALLTRIM(curMailing.First_Name) + ;
                 ALLTRIM(curMailing.Last_Name) + ;
                 ALLTRIM(STR(curMailing.Contact_Id)) + ".pdf"
  lcOutputFile = ADDBS(SYS(2023)) + lcFileName

  * Generate the PDF file
  SELECT curList
  loPDF.PrintReport("ContactListing", lcOutputFile)

  loIPMail.cMailServer  = ALLTRIM(emailinfo.cMailServe)
  loIPMail.cSenderEmail = ALLTRIM(emailinfo.cSender)
  loIPMail.cSenderName  = ALLTRIM(emailinfo.cSenderName)

  loIPMail.cRecipient   = ALLTRIM(curMailing.Email_Name)
  loIPMail.cSubject     = ALLTRIM(emailinfo.cSubject)
  loIPMail.cMessage     = ALLTRIM(emailinfo.cMessage) + ;
                          ALLTRIM(emailinfo.cSignature)

  * Here is where we attach the PDF file
```

```
  IF FILE(lcOutputFile)
    loIPMail.cAttachment  = lcOutputFile
  ENDIF

  lcSentMsg = "To: " + loIPMail.cRecipient + ;
              CHR(13) + "From: " + loIPMail.cSenderEmail + ;
              IIF(EMPTY(loIPMail.cCCList), SPACE(0), CHR(13) + "CC: " + ;
                                          loIPMail.cCCList) + ;
              IIF(EMPTY(loIPMail.cBCCList), SPACE(0), CHR(13) + "BCC: " + ;
                                           loIPMail.cBCCList) + ;
              CHR(13) + "Subject: " + loIPMail.cSubject + ;
              CHR(13) + loIPMail.cMessage

  * Only send the list of produced
  IF FILE(lcOutputFile)
    * Send only if passing parameter, allows testing
    * without sending the email
    IF tlEmail
      llResult = loIPMail.SendMail()
    ELSE
      llResult = .F.
    ENDIF
  ELSE
    llResult = .F.
  ENDIF

  IF !llResult
    WAIT WINDOW "No email message to " + loIPMail.cRecipient + " (" + ;
                loIPMail.cErrorMsg + ")" NOWAIT
    lcSentMsg = lcSentMsg + CHR(13) + CHR(13) + ;
                IIF(tlEmail, "Intended to email", "Not intended to email") +;
                CHR(13) + ;
                "ERROR: " + loIPMail.cErrorMsg

    INSERT INTO emailhistory (tTimeStamp, lSentEmail, mMessage, cRecipient) ;
      VALUES (DATETIME(), .F., lcSentMsg, curMailing.Email_Name)
  ELSE
    WAIT WINDOW "Sent message to " + loIPMail.cRecipient NOWAIT
    lcSentMsg = lcSentMsg + CHR(13) + CHR(13) + "Message sent successfully"

    INSERT INTO emailhistory (tTimeStamp, lSentEmail, mMessage, cRecipient) ;
      VALUES (DATETIME(), .T., lcSentMsg, curMailing.Email_Name)
  ENDIF
ENDSCAN

loPDF    = .NULL.
loIPMail = .NULL.

USE IN (SELECT("curMailing"))
USE IN (SELECT("curList"))
USE IN (SELECT("emailhistory"))
USE IN (SELECT("emailinfo"))
USE IN (SELECT("contacts"))

RETURN
```

##WEB ICON

The example code list is only a partial list of the code in the example program. The wwIPStuff included in the chapter downloads is a shareware version that is available on the West Wind website (www.west-winds.com). It demonstrates the simple implementation of the wwIPStuff class and corresponding DLL file, which are included in Web Connect, or can be purchased separately. The shareware version will display a WAIT WINDOW, but allows complete concept/prototype testing before purchasing the commercial product.

The base idea is to generate the PDF file and attach it to an email. Since this implementation directly sends the email via Simple Mail Transfer Protocol (SMTP), it bypasses all email clients. This means that there will be no audit trail of the sent mail item in a Sent Item folder. While it is nice to trust that the email is safely transferred via the Internet, our customers like to have a record that the email was sent and some details to what was included. The second half of the program provides a basic audit trail of the email, if it was sent successfully, and if not, what error occurred.

To test this program out you will need to change a few columns in the EmailInfo table. The cMailServer is the SMTP server for your email account, cSender is your email address, cSenderName is your name, cMessage is the narrative contents of the message in the email, and cSignature allows for an optional signature line for the message.

We set up the program with a parameter (tlEmail) so the program can be run without actually sending the email. If you run this program with the parameter set to .T., please change the email addresses in the Contacts table to something you will receive and not the chapter author and his partners.

How can I replace the Visual FoxPro Report print preview? *(Example: AltPreview.scx)*

If you poll Visual FoxPro developers and have them note one weakness in Visual FoxPro, my guess is that a big percentage of them would point to the Report Designer Preview mode. It has not had a major enhancement since the days of version 2.x. There are plenty of issues with the display depending on the printer drivers, video drivers and the monitor resolution. The zoom feature has limited percentage settings. It has no drill down capability and shows its age by not displaying hyperlinks. One day I thought, why not use Acrobat to act as the report print preview instead of the standard Visual FoxPro method?

Previously in this chapter we demonstrated a method to generate the PDF file without user interaction. Now all we need is a method of displaying the document in the Acrobat Reader. Not a problem, the following line of code works just fine on our PC:

```
RUN /n1 ;
    "C:\Program Files\Adobe\Acrobat 5.0\Acrobat\Acrobat.exe" ;
    "C:\My Documents\MemberList200008.PDF"
```

So now we need a way to make the call generic. There are several solutions to this. We can store the location in a configuration table or INI file. While this works it is just one more thing that the users need to maintain and can possibly set up wrong, which potentially will lead to another support call. So how can you determine the location of Acrobat? Fortunately,

Acrobat registers itself in the Windows registry and the executable is stored in several keys. The key that seems appropriate for this exercise is:

```
[HKEY_CLASSES_ROOT\AcroExch.Document\shell\print\command]
```

The results will differ based on which version of Acrobat is installed, full product or just the Reader and the OS platform you are using. It is important to note that you will need the full product to generate the PDF files to start with unless you have a product like the Amyuni PDF Converter. On our computers the registry entry consists of the following values:

Acrobat (full):
C:\Program Files\Adobe\Acrobat 5.0\Acrobat\Acrobat.exe

Acrobat Reader:
C:\Program Files\Adobe\Acrobat 5.0\Reader\AcroRd32.exe

So with this functionality we can now use a registry class to grab the location of the executable. The example created (AltRptPreview.scx:: RptPreview() method) will use the same technique as the Acrobat hands free example (including the wwPDF50 classes from Rick Strahl). It uses the Registry class that comes as part of the Fox Foundation Classes (FFC) to determine the location of Acrobat and executes the reader with the PDF file as the parameter.

```
lcRegFile = HOME(2)+"classes\registry.prg"
lcAppKey  = ""
lcAppName = ""
loPDF     = CREATEOBJECT('wwPDF40')

* Check for the existence of the registry class
IF NOT FILE(lcRegFile)
  MESSAGEBOX("Registry class was not found (" + lcRegFile + ")")
  RETURN
ENDIF

* Instance the Registry object
loReg     = NEWOBJECT("FileReg", lcRegFile)

* Get Application path and executable
lnErrNum  = loReg.GetAppPath("PDF", @lcAppKey, @lcAppName)

IF lnErrNum != 0
  MESSAGEBOX("No information available for Acrobat application.")
  RETURN
ENDIF

* Remove switches here (i.e., C:\EXCEL\EXCEL.EXE /e)
IF ATC(".EXE", lcAppName) # 0
  lcAppName = ALLTRIM(SUBSTR(lcAppName, 1, ATC(".EXE", lcAppName) + 3))

  IF ASC(LEFT(lcAppName, 1)) = 34    && check for long file name in quotes
    lcAppName = SUBSTR(lcAppName, 2)
  ENDIF
ENDIF
```

Now that you have the location of the Acrobat executable you can proceed with the building of the file and shell out to Acrobat in "preview mode".

```
* Build the file name for the PDF
lcFileName   = "ContactList" + lcNow + ".pdf"
lcOutputFile = ADDBS(SYS(2023)) + lcFileName

* Generate the PDF file
loPDF.PrintReport("ContactListing", lcOutputFile)

* Run Acrobat or Acrobat Reader
RUN /n1 ;
    &lcAppName ;
    &lcOutputFile
```

The RUN command does not wait for the Acrobat application to be shut down. This is important in the fact that any code that follows the preview will execute. Therefore do not run code to clean up the PDF file because they are open.

##NOTE ICON

It should be noted that repeated calls to run any version of Acrobat will open up another PDF file in the one single instance of Acrobat. This has no effects on the ability for the user to review any of the files. As with anything in the computing world, the limits are memory, file handles, and other system resources.

Another way to do this is:

```
* Example call:
DO shell WITH "ContactListing.PDF", ;
              "C:\My Documents\", ;
              "open"

* Program  : Shell.prg
*   WinApi : ShellExecute
*  Function: Opens a file in the application
*            that it's associated with.
*      Pass: lcFileName -  Name of the file to open
*
*    Return:  2 - Bad Association (ie, invalid URL)
*            31 - No application association
*            29 - Failure to load application
*            30 - Application is busy
*
*            Values over 32 indicate success
*            and return an instance handle for
*            the application started (the browser)
LPARAMETERS tcFileName, tcWorkDir, tcOperation

LOCAL lcFileName, ;
      lcWorkDir, ;
      lcOperation

IF EMPTY(tcFileName)
  RETURN -1
ENDIF
```

```
lcFileName  = ALLTRIM(tcFileName)
lcWorkDir   = IIF(TYPE("tcWorkDir") = "C", ;
                     ALLTRIM(tcWorkDir),"")
lcOperation = IIF(TYPE("tcOperation")="C" AND ;
                  NOT  EMPTY(tcOperation), ;
                  ALLTRIM(tcOperation),"Open")

* ShellExecute(hwnd, lpszOp, lpszFile, lpszParams,;
*              lpszDir, wShowCmd)
*
* HWND hwnd         - handle of parent window
* LPCTSTR lpszOp    - address of string for operation to perform
* LPCTSTR lpszFile  - address of string for filename
* LPTSTR lpszParams - address of string for executable-file parameters
* LPCTSTR lpszDir   - address of string for default directory
* INT wShowCmd      - whether file is shown when opened
DECLARE INTEGER ShellExecute ;
        IN SHELL32.DLL ;
        INTEGER nWinHandle,;
        STRING cOperation,;
        STRING cFileName,;
        STRING cParameters,;
        STRING cDirectory,;
        INTEGER nShowWindow

RETURN ShellExecute(0,lcOperation,lcFilename, SPACE(0), lcWorkDir,1)
```

So what are some of the advantages of this reporting alternative? In our opinion, it addresses some of the Visual FoxPro Report Writer drawbacks. It mainly addresses the weakness of the preview zoom (or as it is really known as, “lack of zoom”). The Acrobat Reader provides super zoom capability (12.5% up to 1600%). Other nice to have features are having multiple pages visible at one time with continuous mode, a search feature, and a true What-You-See-Is-What-You-Get (WYSIWYG). You can also view multiple PDF reports since the Acrobat Reader can open multiple PDF files.

Visual FoxPro developers have been challenged by the Visual FoxPro Report Designer and have not been bashful about voicing these issues. Microsoft has repeatedly noted that there will be little to nothing addressed with the existing Report Designer in future versions of Visual FoxPro. Microsoft has also noted that we live in a component world. This is a beautiful example of that component world reaping benefits for our clients. The example uses Rick Strahls wwPDF class to avoid the user interaction when the PDF file is generated before it is previewed in Acrobat. The code can be altered to use any one of the other PDF generators that are available to developers.

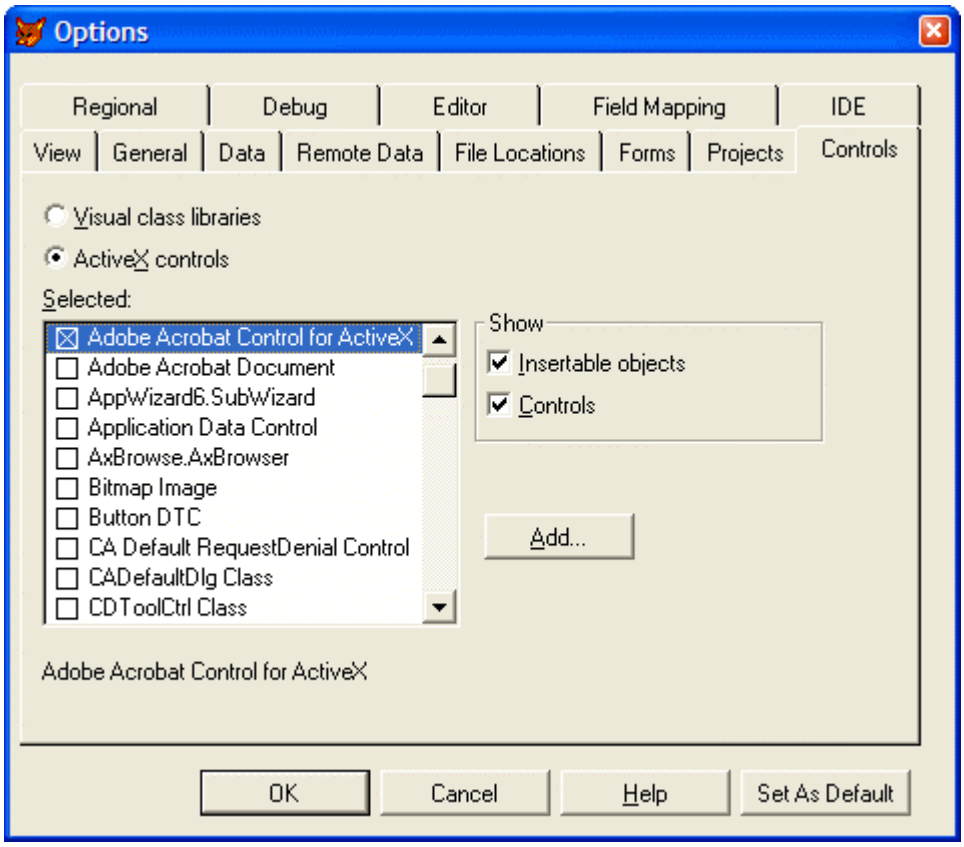
How do I present Acrobat PDFs in a Visual FoxPro Form?

(Example: PdfDisplay5.scx, PdfDisplay5a.scx)

If you have Acrobat or the Acrobat Reader product you will also have the ActiveX control that will display a PDF file in a Visual FoxPro form. There are two controls that appear in the Tools |Options dialog on the Controls page. The control you want to work with is Acrobat control for ActiveX. The other control, Adobe Acrobat document only allows you to hard code the PDF file that is displayed.

We want to give you a word of caution before moving into development with this control. We originally developed the samples with the control included in Acrobat 4.0. These samples have worked flawlessly. In March of 2001 Acrobat 5.0 version was release. We have crashed Visual FoxPro 6 and Visual FoxPro 7 a number of times with the newest version. The examples presented have worked around the C5 errors. Adobe states specifically on their website that this control was designed specifically to work with Microsoft's Internet Explorer, yet discusses its use with developer tools like Visual Basic. So tread carefully with the examples and implementation in applications.

Like all ActiveX controls, first you will need to select the Acrobat Control for ActiveX in the Controls tab of the Visual FoxPro Options dialog.



##IMAGE: MF08005.tif

Figure 8.5 *The Acrobat Control for ActiveX is available in the Controls tab of the Visual FoxPro Options dialog.*

Building the form is straightforward. Drop the control from the ActiveX palette on the Visual FoxPro Form Controls toolbar to a Visual FoxPro form.

##IMAGE: MF08006.tif

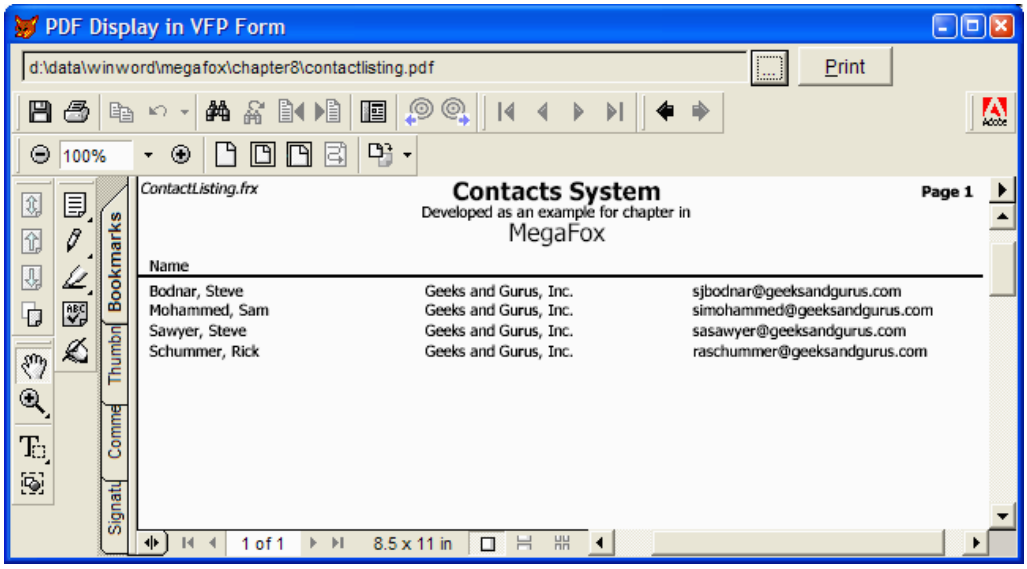
Figure 8.6 *The Acrobat Control is the middle toolbar button (with Acrobat symbol).*

The property that needs to be set and/or bound to a Visual FoxPro control is SRC. This tells the Acrobat control which PDF file to load and display. The SRC property can be set dynamically which reloads the selected PDF file in the viewer (this worked fine in Acrobat 4 and causes OLE errors in Acrobat 5 unless set in the form Init method). The example form (PdfDisplay5.scx, included in the downloads available from www.Hentzenwerke.com) takes a parameter, which is the PDF file name and sets the SRC property of the PDF ActiveX control.

```
* PdfDisplay5.scx Init()
LPARAMETERS tcPdfFileName

this.Resize()

IF VARTYPE(tcPdfFileName) = "C" AND FILE(FULLPATH(tcPdfFileName))
  this.olePDF.SRC = FULLPATH(tcPdfFileName)
ELSE
  this.olePDF.SRC = FULLPATH(this.olePDF.SRC)
ENDIF

this.olePDF.setFocus()
this.olePDF.setZoom(150)

RETURN
```

The sample form has a couple of things you should note before trying to run it. The first is that you must have the ActiveX control registered on your PC. The second is that we have hard coded the PDF filename in the SRC property. There is a good chance that your directory structure does not match ours so some changes will need to be implemented before running the form or you will need to pass in the parameter, which is the PDF file name (fully pathed or available on the Visual FoxPro path). If the PDF is not available the form is displayed empty since Acrobat cannot load the PDF.

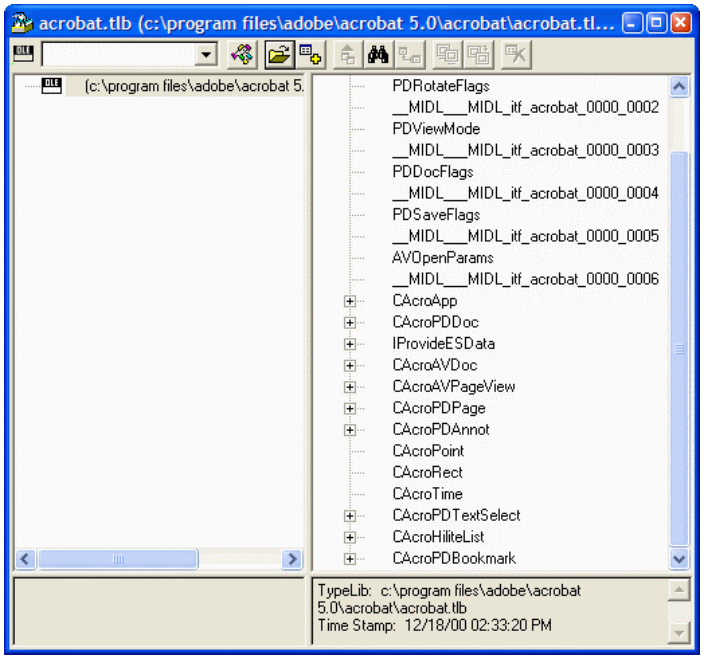
The form is displayed (see Figure 8.7.) with the PDF visible. Do not be surprised by the Acrobat splash screen. This is displayed when the Acrobat ActiveX control is instanced (the same behavior is displayed when a PDF file is opened in Internet Explorer). All of the toolbars that are included in the Acrobat Reader (or full version if this is what is loaded on the PC) are available in your Visual FoxPro form including tools to zoom in and out, print the document, search for text, change pages, and save it off to another file. Even items like Bookmarks and Thumbnails are available in the ActiveX control.



##IMAGE: MF08007.tif

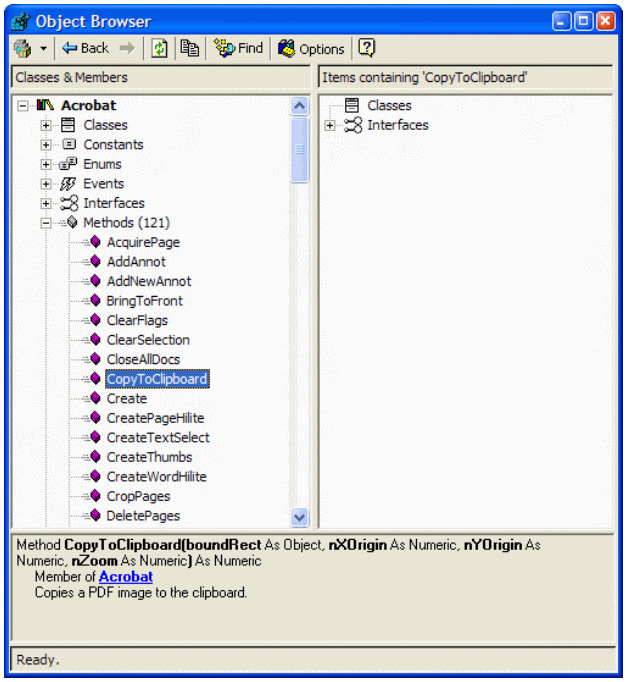
Figure 8.7 *This is a PDF file displayed in a Visual FoxPro form. The Print command button will display the printer selection dialog for the user.*

There are a number of methods that can be called to change the behavior of the PDF viewer. Unfortunately there is no documentation in the ActiveX control properties dialog that describes the method parameters nor is there an associate help file. We can open up the ActiveX control (PDF.OCX) or the control's typelib file (PDF.TLB) to see what the parameters are. Still, there is no specific documentation that we could find before assembling this chapter. In Visual FoxPro 6 you need to use the Class Browser (see Figure 8.8), in Visual FoxPro 7 you will need to use the new Object Browser (see Figure 8.9).



##IMAGE: MF08008.tif

Figure 8.8 *The Acrobat Control for ActiveX exposes a number of methods for the developer to interact with the control in the Visual FoxPro form. This is the exposed in the Visual FoxPro 6 Class Browser.*



##IMAGE: MF08009.tif

Figure 8.9 *To view the property, events and methods in Visual FoxPro 7 you need to use the Object Browser.*

All of the features you use in Acrobat Reader via the menus and toolbars are exposed in the Acrobat ActiveX Control. There may be methods that might be handy to execute via your own exposed interface. A number of the Reader features are exposed through an interface of properties and methods. Note the method names usually start out with a lower case name (visible in the Object Browser and the Acrobat Javascript documentation). This is due to the standard that Javascript uses, which is the native "macro" language included in Acrobat.

The printWithDialog() is nice because it automatically displays the printer selection and print driver option dialog that Acrobat displays when you select the File | Print menu option. You can also print directly to the Windows' default printer with the Print() method. There are a number of print methods to suit most tastes. The gotoLastPage() method could be used in the cases when the customer likes to view the grand total information on the report which is on the last page, before reviewing the details. If your users prefer to see the report zoomed at a specific percentage you can use the setZoom() method.

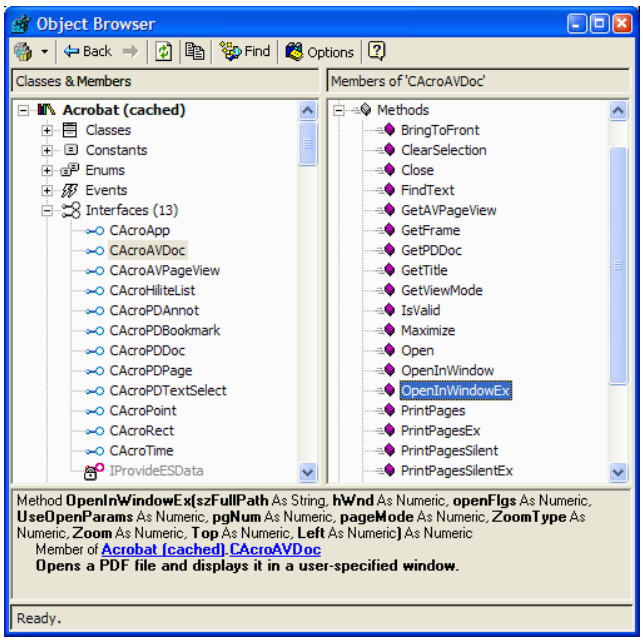
There is a second method to displaying PDFs in a Visual FoxPro form. If you have the full Acrobat product you will also have the ActiveX interface that will display a PDF file in a VFP form. This interface is **not** loaded with the Reader edition of Acrobat. However this object is well documented, both in the type library and in the Acrobat Software Developers Kit (SDK).

##WEB ICON

The Acrobat Software Developers Kit can be downloaded from the Adobe developer page located at http://partners.adobe.com/asn/developer/acrosdk/acrobat.html.

The easiest way to work with this technique is to use the new VFP 7 Object Browser. Open up the Acrobat 5.0 Type Library object (see Figure 8.10). The main object is located under Interfaces. It is called CAcroAVDoc. This interface has the capability to get at the other needed interfaces as well as display the PDF. This object is created in the *Init* of the form.

```
* Form Init()
LPARAMETERS tcPDF

IF DODEFAULT(tcPDF)
  this.oAVDoc = CREATEOBJECT("AcroExch.AVDoc")
  this.Navigate(tcPDF)
ENDIF

RETURN
```



##IMAGE: MF08014.tif

Figure 8.10 *The Acrobat 5.0 Type Library object has documented interfaces, method, and constants.*

The form will optionally accept a PDF file name as a parameter. If the Acrobat object cannot be instantiated the *Error* method will trap the condition and disable the user interface

objects on the form. After the object is created the custom Navigate method is called to open up and display the PDF file. To open the PDF file and display it in the form we use the new Visual FoxPro *Hwnd* property as a parameter to the OpenInWindowEx method. This allows Acrobat to display itself in a Visual FoxPro form.

```
* Form Navigate()
LPARAMETERS tcPDF

* Constants extracted from Acrobat 5.0 Type Library via Object Browser

#DEFINE AVZoomNoVary            0      && Fixed value zoom.
#DEFINE AVZoomFitPage           1      && Fit page to window.
#DEFINE AVZoomFitWidth          2      && Fit page width to window.
#DEFINE AVZoomFitHeight         3      && Fit page height to window.
#DEFINE AVZoomFitVisibleWidth   4      && Fit visible width to window.
#DEFINE AVZoomPreferred         5      && Use page's preferred zoom.
#DEFINE pdRotate0               0      && Rotated 0 degrees.
#DEFINE pdRotate90             90      && Rotated 90 degrees.
#DEFINE pdRotate180           180      && Rotated 180 degrees.
#DEFINE pdRotate270           270      && Rotated 270 degrees.
#DEFINE PDDontCare              0      && Leave the view mode as it is.
#DEFINE PDUseNone               1      && Display the document without
bookmarks or thumbnails.
#DEFINE PDUseThumbs             2      && Display the document and thumbnail
images.
#DEFINE PDUseBookmarks          3      && Display the document and bookmarks.
#DEFINE PDFullScreen            4      && Display the document in full screen
mode.
#DEFINE PDDocNeedsSave          1      && Document has been modified and needs
to be saved.
#DEFINE PDDocRequiresFullSave   2      && Document cannot be saved
incrementally; it must be written using PDSaveFull.
#DEFINE PDDocIsModified         4      && Document has been modified.
#DEFINE PDDocDeleteOnClose      8      && Document is based on a temporary
file.
#DEFINE PDDocWasRepaired       16      && Document was repaired when it was
opened.
#DEFINE PDDocNewMajorVersion   32      && Document's major version is newer
than current.
#DEFINE PDDocNewMinorVersion   64      && Document's minor version is newer
than current.
#DEFINE PDDocOldVersion       128      && Document's version is older than
current.
#DEFINE PDDocSuppressErrors   256      && Don't display errors.
#DEFINE PDDocIsEmbedded       512      && Document is embedded in a compound
document.
#DEFINE PDDocIsLinearized    1024      && Document is linearized (get only).
#DEFINE PDDocIsOptimized     2048      && Document is optimized.
#DEFINE PDSaveIncremental       0      && Write changes only.
#DEFINE PDSaveFull              1      && Write the entire file.
#DEFINE PDSaveCopy              2      && Write a copy of the file into the
file.
#DEFINE PDSaveLinearized        4      && Save the file in a linearized
fashion.
#DEFINE PDSaveWithPSHeader      8      && Writes a PostScript header as part of
the saved file.
#DEFINE PDSaveBinaryOK         16      && Specifies that it's OK to store in
binary file.
```

```
#DEFINE PDSaveCollectGarbage   32      && Remove unreferenced objects, often
reducing file size.
#DEFINE AV_EXTERNAL_VIEW        1      && Open the document with the tool bar
visible.
#DEFINE AV_DOC_VIEW             2      && Draw the page pane and scrollbars.
#DEFINE AV_PAGE_VIEW            4      && Draw only the page pane.

IF VARTYPE(tcPDF) = "C"
  IF FILE(tcPDF)
    WITH this.oAVDoc
      * It's important to close each doc, every time. If you don't, when you
      * try viewing the same page, it won't display anything - you have to kill
      * the object references and close VFP, plus kill Adobe. It maintains a
      * collection of open documents , but we are only using one document at a
      * and the zero makes sure the document is not saved. It keeps things
simple,
      * and to keep the memory usage to a minimum.
      .Close(0)

      .OpenInWindowEx(tcPDF, this.Hwnd, AV_EXTERNAL_VIEW, ;
                      .T., 0, PDUseNone, AVZoomPreferred, ;
                      100 , 30, 0)

      this.oAVPage  = .GetAVPageView()

      * Set the zoom options
      this.ResizeAcrobat()

      IF !ISNULL(this.oAVPage)
        * Turn on, preset the zoom control on the form. Then zoom to the
        * correct PDF size.
        this.oAVPage.ZoomTo(0, 100)
        this.oAvPDDoc = .GetPDDoc()
      ENDIF

      this.cOpenPDF = this.FormatFileName(tcPDF)
      this.Refresh()
    ENDWITH
  ELSE
    MESSAGEBOX("PDF File selected does not exist", ;
               0 + 64, this.Caption)
  ENDIF
ENDIF

RETURN
```

We decided to include all the #DEFINEs so you can see the various options available. The Navigate method first closes an existing PDF if one is open, then opens up the selected PDF and displays it. The Navigate method also instantiates two more Acrobat objects. The first is based on the CAcroAVPageView interface. There are a number of methods available on this object to manipulate to a specific location in the document and determine what the user will see. Methods include ScrollTo (to scroll to a specific location on a page), ZoomTo (to zoom the document to a certain percentage), DoGoBack (to return to the previous position in the view history stack), and DoGoForward (to return to the next view in the history stack). The second is based on the CAcroPDDoc interface. This object provides methods GetNumPages (to find out the number of pages, handy when printing the documents or ranges of pages),

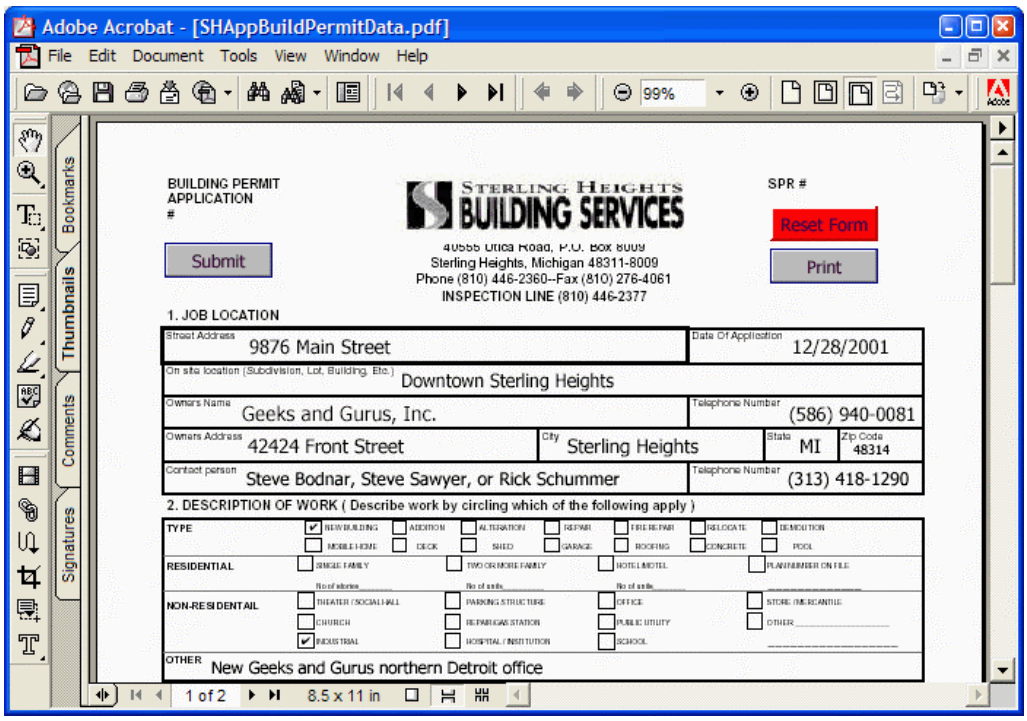
GetFileName (to know what PDF is open), DeletePages/CreateThumbs/DeleteThumbs (if you want to manipulate the contents of the documents), and Save (does what you would expect). We did not implement all of these methods, we thought that it would take all the fun away from you and left that as an exercise for you to become familiar with the different objects.

The sample form will open up without showing a PDF if you do not pass the file as a parameter. The user can then use the ellipses button (three dots) to select a PDF. This button uses the **`GETFILE`** function to obtain the PDF name then calls the Navigate method. The user can resize it and have the PDF viewer resize itself as well. The only real drawback of this technique is that it is only available to users that have the full version of Acrobat installed.

What is Acrobat Forms Author technology? *(Example: SHAppBuildPermitData.pdf)*

Acrobat ships with a cool feature called Acrobat Forms Author Technology that is provided via a plug-in (add-on or extension to the base product). This technology allows end users to convert paper forms into electronic forms that have the exact look of the original paper forms. These forms can be displayed in Acrobat and the end users can enter data in the same exact format they used when filling out the paper directly. This form might be a company standard, an industry directive, or a governmental dictate.

If your users are as demanding as ours, you have probably run into the situation where you have been asked to produce an interface form that duplicates the existing paper version. You go off to develop this slick interface and demo the prototype to the users. The first thing they mention is that it does not mimic the paper version of the form “exactly”. The flip side is printing reports that mimic the paper version. While this is usually easier than the data entry part of the equation, generating reports with various lines and boxes, detail lines that exceed the facilities of the Visual FoxPro Report Designer or even some of the third party report writers can be a challenge. Once and awhile it is impossible. Acrobat Forms can assist us in getting data via data entry and outputting data to the forms for printing.



##IMAGE MF08010.tif

Figure 8.11 *This form is the city of Sterling Heights Building Permit form with some data filled in as the user would see it in Acrobat.*

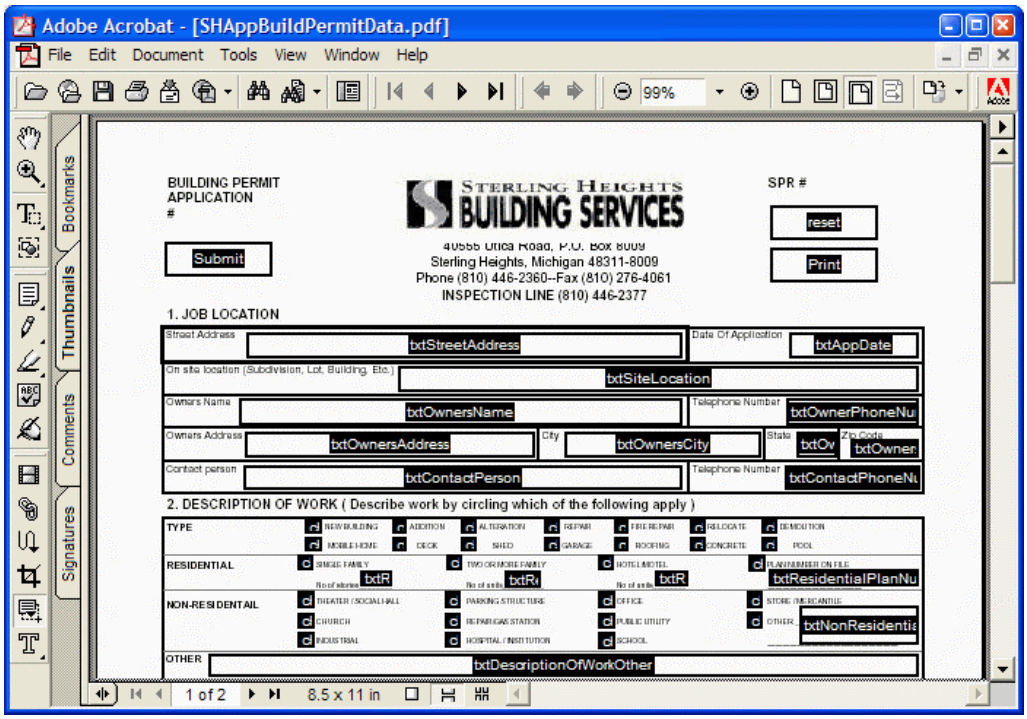
The Forms Author plug-in capability is included with the full version of Acrobat. The data entry mode is available in the Reader version as well as full Acrobat, and a new product called Acrobat Approval. So what are the advantages? For one, the forms can be replicated electronically just like they are on paper. Since Acrobat printing is truly What-You-See-Is-What-You-Get (WYSIWYG) the forms can be printed after being filled in. They can be saved with the data entered, which provides an audit trail. Most importantly, the information can be extracted and saved in a database for further analysis.

Visual FoxPro developers might be asking the question, why would I need Acrobat forms when I have a great forms designer in Visual FoxPro? The difference is that Acrobat Forms can also be implemented in a distributed environment via the Internet without the overhead of the ActiveDoc technology used in Visual FoxPro. This means that the PDF file can be accessed on the web, users can enter in data, and the information can be submitted to the web server for processing and the data extracted and stored into a database.

There are a couple of concepts in developing these forms that are very familiar to Visual FoxPro developers. The Acrobat Forms “designer” has similar functionality as the Visual FoxPro form designer. You change the mode of Acrobat with the PDF from “entry” to “designer” via the Form Tool icon on the left-side toolbar icon (second from the bottom on the left side toolbar in Figure 8.12). This toggles the mode so the form editor is available. Right-

click on any object will bring up the shortcut menu. One of the many menu options is Properties. Selecting this option will introduce the Acrobat form field property sheet (see Figure 8.13 for one page in this dialog). There are properties to name the objects, comment their use, adjust fonts, format the entry, set colors, require data, make it read only, have default values, set the tab order, and align the text. There are settings to run code for events and perform validation. Sound familiar? Sounds like what we do with the Visual FoxPro Class and Form Designers on a regular basis. Object types include Text (TextBox), CheckBox, ComboBox, ListBox, RadioButton (OptionGroup), Button (CommandButton) and Signature (no Visual FoxPro equivalent).

The property dialog is very comfortable to Visual FoxPro developers. The biggest difference is that the code is written in JavaScript. Dropping objects on the PDF form is completed by changing the PDF into "designer mode" as noted earlier, and clicking and dragging to size the new object. This will open the field properties dialog. You select the object type and start setting the various properties. Each subsequent time you drag on another object it will default to the same object type as the previous one added.

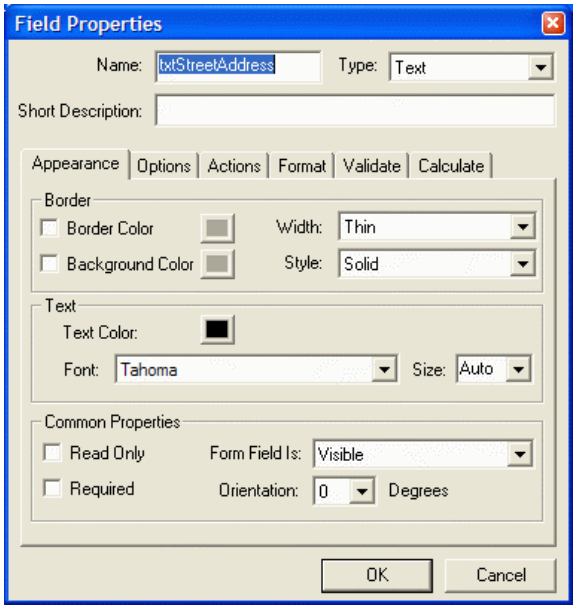


##IMAGE MF08011.tif

Figure 8.12 *This is the same form, but now seen in "designer mode".*

Implementation of a PDF with Forms is identical to a regular PDF file. These files can be opened, data entered, forms printed with the Reader version of Acrobat. The PDF can also be

saved with the data included using the full version of Acrobat. The Amyuni product does not have any functionality concerning Acrobat Forms.



##IMAGE MF08012.tif

Figure 8.13 *This is the Appearance page on the Acrobat Form Object Property Sheet for a Text object.*

To this point we have not discussed the interaction with Visual FoxPro. The data captured in an Acrobat Form is exported via the File | Export | Form Data… menu option. This option is only available with the Business Tools or full Acrobat editions in version 4 of Acrobat. Version 5 requires the Approval or full Acrobat version. The export process creates a FDF file. This file is a flat text file that includes tags and data. Here is the information in the FDF file as it was exported from the SHBuildPermitData.pdf (included with downloads):

```
%FDF-1.2
%âãÏÓ
1 0 obj
<<
/FDF << /Fields [ << /V /Off /T (chkNonResidentialTheater)>> << /V /Off /T
(chkResidentialChurch)>>
<< /V /Off /T (chkResidentialGasStation)>> << /V /Off /T
(chkResidentialHospital)>>
<< /V /Off /T (chkResidentialHotelMotel)>> << /V /Yes /T
(chkResidentialIndustrial)>>
<< /V /Off /T (chkResidentialOffice)>> << /V /Off /T (chkResidentialOther)>>
<< /V /Off /T (chkResidentialParkingStructure)>> << /V /Off /T
(chkResidentialPlanNumberOnFile)>>
<< /V /Off /T (chkResidentialPublicUtility)>> << /V /Off /T
(chkResidentialSchool)>>
```

```
<< /V /Off /T (chkResidentialSingle)>> << /V /Off /T (chkResidentialStore)>>
<< /V /Off /T (chkResidentialTwoOrMore)>> << /V /Off /T (chkTypeAddition)>>
<< /V /Off /T (chkTypeAlteration)>> << /V /Off /T (chkTypeConcrete)>>
<< /V /Off /T (chkTypeDeck)>> << /V /Off /T (chkTypeDemolition)>>
<< /V /Off /T (chkTypeFireRepair)>> << /V /Off /T (chkTypeGarage)>>
<< /V /Off /T (chktypeMobileHome)>> << /V /Yes /T (chkTypeNewBuilding)>>
<< /V /Off /T (chkTypePool)>> << /V /Off /T (chkTypeRelocate)>>
<< /V /Off /T (chkTypeRepair)>> << /V /Off /T (chkTypeRoofing)>>
<< /V /Off /T (chkTypeShed)>> << /V (11/15/2001)/T (txtAppDate)>>
<< /V (12/31/2099)/T (txtContactorHomeExpirationDate)>> << /V (38-9999999)/T
(txtContactorHomeFedId)>>
<< /V (987654321)/T (txtContactorHomeLicenseNumber)>> << /V (VFP Specialists)/T
(txtContactorHomeLicenseType)>>
<< /V (Sterling Heights)/T (txtContactorHomeOwnerCity)>> << /V (48313)/T
(txtContactorHomePostalCode)>>
<< /V (MI)/T (txtContactorHomeState)>> << /V (5865551234)/T
(txtContactorHomeTelephoneNumber)>>
<< /V (Weasel and Shifty Insurance Group, Inc.)/T
(txtContactorHomeWorkerCompIns)>>
<< /V (Steve Bodnar, Steve Sawyer, or Rick Schummer)/T (txtContactPerson)>>
<< /V (3134181290)/T (txtContactPhoneNumber)>> << /V (Acme Construction)/T
(txtContractorHomeOwner)>>
<< /V (9999 Elms Street)/T (txtContractorHomeOwnerAddress)>> << /V (New Geeks
and Gurus norther Detroit office)/T (txtDescriptionOfWorkOther)>>
<< /V (D3726312873621878)/T (txtDriverLicense)>> << /V (999999999999)/T
(txtMESC)>>
<< /V (5869400081)/T (txtOwnerPhoneNumber)>> << /V (42424 Front Street)/T
(txtOwnersAddress)>>
<< /V (Sterling Heights)/T (txtOwnersCity)>> << /V (Geeks and Gurus, Inc.)/T
(txtOwnersName)>>
<< /V (48314)/T (txtOwnersPostalcode)>> << /V (MI)/T (txtOwnersState)>>
<< /V (Downtown Sterling Heights)/T (txtSiteLocation)>> << /V (9876 Main
Street)/T (txtStreetAddress)>>
]
/F (SHAppBuildPermitData.pdf)/ID [
<8c562dff8dbc2284ab14a9e4b572b02f><98995e30afea0090038a1c9c79587e1d>
] >>
>>
endobj
trailer
<<
/Root 1 0 R

>>
%%EOF
```

At this point we can see that the data entered can be output to a flat file. This file can be parsed using Visual FoxPro's Low-Level File Input and Output commands and added to tables which are much easier for us to process. It would require that some fundamentally mundane code be written to separate the information from the tags and to get this information into a table. While most of us would not mind writing this code, wouldn't it be cool if there was a better mechanism to extract the data from the FDF format? There is and it is called the FDF Toolkit, from Adobe.

How can I extract data out of a PDF form file? *(Example: FDFRead.prg)*

So now that we understand Acrobat PDF files can be built as a data entry mechanism and provide printing capability, the question begs, how do we extract this data from an Acrobat form and have it interact with our custom database applications? Adobe has provided a product called the FDF Toolkit on their website (http://partners.adobe.com/asn/developer/acrosdk/forms.html). This is a free product with a version for Acrobat 4 and 5 (our experience is that the version for 4 works with Acrobat 5, it just has fewer features). The download includes Application Programming Interfaces (API) for C/C++, Java, Perl, and ActiveX, and some extensive documentation on how it can be used with these tools. Visual FoxPro developers will find the Win32 ActiveX interface of the FDF Toolkit easy to use and very compatible (despite the lack of Visual FoxPro examples in the documentation). The ActiveX portion of the toolkit is made up of two files: FdfAcX.dll and FdfTk.dll. The toolkit will install the toolkit files, but does not register the components.

The examples to read and write a FDF file will seem very familiar if you have worked with any Automation to Microsoft Word and the Visual FoxPro Low Level File Input/Ouput commands (LLFIO). The example code can be found in the FDFRead.prg and the FDFWrite.prg samples which can be downloaded from Hentzenwerke.

Register the FDF Toolkit ActiveX control

The ActiveX control (FdfAcX.dll and corresponding FdfTk.dll) should reside in the Windows/System32 directory or another directory that has "execute" permission. The process to register the FDF Toolkit ActiveX control is as simple as the following command (add a path to the DLL if necessary):

```
RegSvr32 FdfAcX.dll
```

The control is self-registering. The Visual FoxPro 6 Setup Wizard and Visual FoxPro 7 InstallShield Express products will automatically register this control as part of the installation process so the deployment process is easy. Please note that there is no reason to register the FdfTk.dll and that it will fail if you try to do so.

Instantiating the object to access the FDF File

The instantiation of the FDF ActiveX interface is accomplished via a standard process of using the Visual FoxPro **CREATEOBJECT()** function. Here is an example of the needed code:

```
loFDF = CREATEOBJECT("fdfApp.FdfApp")
```

This returns an object reference to the FDF control so that the methods can be run to read and write data from the FDF file. Now that we have the important object reference to the FDF control we can start to manipulate the data inside of it via the interface methods that are exposed.

The first step in reading the information is to open the FDF file. This is accomplished by running the FDFOpenFromFile() method.

```
loFDFFile = loFDF.FDFOpenFromFile("SHAppBuildPermitData.fdf")
```

This method returns an object reference to the FDF file. If the file does not exist or could not be opened, an OLE Exception is thrown. You will need to handle this issue in your error-handling scheme. Once the object reference is gained you can go after specific fields in the FDF. To take this approach you need to provide the field name as a parameter to the FDFGetValue() method. One important item to note is the field names in the FDF and access to these fields is case sensitive. The passing of "txtstreetaddress" is not the same as "txtStreetAddress". So, to access a specific field you can use code like:

```
lcFDFField   = "txtStreetAddress"
luFieldValue = loFDFFile.FDFGetValue(lcFDFField)
```

You can also use the FDFNextFieldName() method to loop through the fields. To get the first field in the file you pass a null string (**SPACE(0)**) as the parameter to the FDFNextFieldName() method. To get the next field in the FDF file you pass the current field. Here is some code that loops through all the fields in the FDF file:

```
IF VARTYPE(loFDFFile) = "O"
  * Get the first field name in the FDF file
  lcFDFField     = loFDFFile.FDFNextFieldName("")
  lnFieldCounter = 1

  CLEAR

  * Loop through the FDF file to get the values
  DO WHILE NOT EMPTY(lcFDFField)
    luFieldValue   = loFDFFile.FDFGetValue(lcFDFField)

    ? str(lnFieldCounter, 6), lcFDFField, ;
      "(", vartype(luFieldValue), ") ==", luFieldValue

    lcFDFField = loFDFFile.FDFNextFieldName(lcFDFField)
    lnFieldCounter = lnFieldCounter + 1
  ENDDO
ENDIF

loFDFFile.FDFClose()
```

The data in the FDF file is strictly character based. If you are moving this data into a table you will likely need to transform the data into the proper data type for the field unless the record is all character fields.

There are hundreds of thousands of paper based forms already pre-built, and a large percentage of these are already scanned and available on the Internet in PDF format. The examples used in this chapter were directly downloaded from the Sterling Heights city website. Many of the governmental and private business entities already have the forms set up in PDF format, and some are already set up with the form fields included. All the object fields were added in the example PDFs in less than 45 minutes. We did not add any JavaScript for serious validation or enforce any business rules in the examples, but it can be done with a little more effort. Leveraging existing PDF forms will save you time, your clients' money, and can make you look like the hero.

These forms can be used in a traditional LAN/Workstation based application as well as the Client/Server arena. The users open up Acrobat Reader and fill in the data in the form and

use the menu to save the data to a predefined directory. Each user will need a full license to Acrobat (unless the new and less expensive Acrobat Approval meets your requirements). They will use the menu since the product does not support the JavaScript code necessary to export the data. In a website configuration the users open up the PDF in the browser and fill in the data. The Reader version (as well as the full version of Acrobat) can submit form data back to the webserver with Javascript. We included a Submit button in the SHAppBuildPermit.pdf example to show the simple JavaScript code needed to submit the data back to the webserver. The data submitted from an Acrobat form is sent to the webserver in the same exact format as the data submitted from an HTML form. This information can be processed by a Common Gateway Interface (CGI) process. We have used WebConnect (from West Winds) to be the CGI process that accepts data from a PDF on the web. The great thing about WebConnect in this situation is that it is extremely fast, and it allows Visual FoxPro developers to leverage their Visual FoxPro skills to provide a powerful solution.

How do I prefill the PDF Form with data? *(Example: FDFWrite.prg)*

Reading the file might be enough excitement for some of our clients, but what if they could also prefill a PDF Form with data from their Visual FoxPro application? The FDF Toolkit control also provides a plethora of methods to write out data into the FDF format. Once the object reference to the FDF ActiveX control is obtained, you execute the FDFCreate() method. This creates the FDF in memory and returns an object reference to this "file". After the file is create, the field name tag (/F) and value tag (/V) are written for each of the fields you want written via the FDFSetValue() method. The example below writes out two fields.

```
loFDFFile      = loFDF.FDFCreate()

* Fill in two fields in the FDF
lcFDFField      = "txtStreetAddress"
lcFDFFieldValue = "1002 MegaFox Demo Street"
luFieldValue    = loFDFFile.FDFSetValue(lcFDFField, lcFDFFieldValue, .F.)

lcFDFField      = "txtOwnersName"
lcFDFFieldValue = "Enter your Name Here"
luFieldValue    = loFDFFile.FDFSetValue(lcFDFField, lcFDFFieldValue, .F.)
```

Naturally the code you will write will include more than a couple of fields. You also need to transform data from the native format to character before storing it in the FDF file. The final method called before closing the file is the FDFSetFile(). This writes out the /F tag, which associates the FDF file with the PDF file the data will be prefilled and display in. When the FDF file is opened it will preload the associated PDF file, and then fill in the fields loaded in the FDF.

```
* Set the name of the PDF associated with the FDF
loFDFFile.FDFSetFile("SHAppBuildPermitForm.pdf")
```

The FDFSaveToFile() physically writes out the FDF data to a file. The file is closed via the FDFClose() method and the object reference should be released.

```
* Write out the file
```

```
loFDFFile.FDFSaveToFile("Chapter08Sample.fdf")
loFDFFile.FDFClose()
```

There are a number of other methods in the FDF ActiveX that provide behaviors you may find useful. There are capabilities to write FDF files to a string, additional tags can be inserted into the file, you can add custom JavaScript, etc.

There are two real life examples using the FDF Toolkit to prefill data in a PDF form that we would like to discuss. The first is to use it as a substitution of the Visual FoxPro Report Designer. Customers are always demanding reports that replicate the paper forms. Some of these reports can be quite challenging using the Visual FoxPro Report Designer or any third-party reporting tool. Since we can see that plugging in data into a PDF can be straightforward, why not take advantage of this technique? Generate the FDF reference, plug in the data, and save it to a temporary file. Using the techniques discussed in the section "How can I replace the Visual FoxPro Report print preview?", you can shell Acrobat Reader for the user to preview the report and they can print it using the Reader interface, or via a button like we included in the SHAppBuildPermit.pdf example. You can also display the PDF file in a Visual FoxPro form and manipulate it via the ActiveX interface.

The second example is to place the PDF on a website or in a custom application for data entry. If there is default data that can be plugged into the PDF form from the application's database, use the FDF Toolkit to plug in the data before the user sees the PDF in the reader. We do this with our Visual FoxPro forms all the time, why should using this interface be different. On the Internet you will return the FDF file to the browser which will instance the Acrobat ActiveX control based on the file association of the FDF. The Acrobat control will request the PDF file from the webserver and the PDF will be displayed with data prefilled in the browser.

How can I merge PDF files together? *(Example: PDFMerger.prg/PDFDirectoryMerger.prg)*

This chapter has demonstrated a number of ways to generate PDF files from Visual FoxPro reports. There are times when merging different reports together into one PDF file is a requirement of the customer. This section will discuss one way to accomplish merging two PDFs together using ActiveX components provided with the full version of Adobe Acrobat and then demonstrate how a complete directory of PDF files can be merged into one.

Acrobat has an ActiveX interface. First you instantiate a reference to the AcroExch.App object and an object reference to AcroExch.PDDoc for each of the PDF files that you want merged together. The Open method of the AcroExch.PDDoc opens the PDF file and establishes an object reference to the PDF. The GetNumPages method returns the number of pages in the PDF. It should be noted that the number of pages in the PDF file returned from the GetNumPages is zero based (starts at zero).

The actual merging of the files happens with the InsertPages method. The first parameter is the page number that you want the merge to start after. Typically you will merge after the last page, but you can insert a PDF anywhere in another PDF. The second parameter is an object reference to the second PDF file via the AcroExch.PDDoc object. The third parameter is the start page. Again the internal page numbers in a PDF file start with zero, so if you want to get the first page you would pass a zero. The fourth parameter is the number of pages to insert. The last parameter indicates if you also want the bookmarks inserted as well.

Listing 8.2 Partial code listing of PDFMerger.prg which demonstrates how to merge two PDF files together.

```
LPARAMETERS tcPDFOne, tcPDFTwo, tcPDFCombined, tlShowAcrobat

#DEFINE  ccSAVEFULL 0x0001

LOCAL loAcrobatExchApp, ;
      loAcrobatExchPDFOne, ;
      loAcrobatExchPDFTwo, ;
      lnLastPage, ;
      lnNumberOfPagesToInsert, ;
      lcOldSafety

lcOldSafety = SET("Safety")
SET SAFETY OFF
ERASE tcPDFCombined
SET SAFETY &lcOldSafety

* Get appropriate references to Acrobat objects
loAcrobatExchApp    = CREATEOBJECT("AcroExch.App")
loAcrobatExchPDFOne = CREATEOBJECT("AcroExch.PDDoc")
loAcrobatExchPDFTwo = CREATEOBJECT("AcroExch.PDDoc")

* Show the Acrobat Exchange window
IF tlShowAcrobat
   loAcrobatExchApp.Show()
ENDIF

* Open the first file in the directory
loAcrobatExchPDFOne.Open(tcPDFOne)

* Get the total pages less one for the last page num [zero based]
lnLastPage = loAcrobatExchPDFOne.GetNumPages() - 1

* Open the file to insert
loAcrobatExchPDFTwo.Open(tcPDFTwo)

* Get the number of pages to insert
lnNumberOfPagesToInsert = loAcrobatExchPDFTwo.GetNumPages()

* Insert the pages
loAcrobatExchPDFOne.InsertPages(lnLastPage, loAcrobatExchPDFTwo, 0, ;
                                lnNumberOfPagesToInsert, .T.)

* Close the document
loAcrobatExchPDFTwo.Close()

* Save the entire document, saved as file passed as third
* parameter to program using SaveFull [0x0001].
loAcrobatExchPDFOne.Save(ccSAVEFULL, tcPDFCombined)

* Close the PDDoc
loAcrobatExchPDFOne.Close()

* Close Acrobat Exchange
loAcrobatExchApp.Exit()

* Need to release the objects
```

```
RELEASE loAcrobatExchPDFTwo
RELEASE loAcrobatExchPDFOne
RELEASE loAcrobatExchApp

WAIT CLEAR

RETURN SPACE(0)
```

##NOTE ICON
Acrobat will not merge secured PDF documents. The result of a merge between one secure PDF document and a non-secure PDF document will be the contents of the non-secure PDF document.

##IMAGE MF08013.tif
Figure 8.14 *The Acrobat Document Security screen (File | Document Security… menu) will inform you of the security settings for the PDF file.*

We have found the performance of the merge functionality to be very snappy. We have merged small PDFs (10 kilobytes) with large PDFs (over 1 megabyte), and large PDFs with other large PDFs in a couple of seconds or less. The merge process will also merge the bookmarks in one or both documents.

The merge process is useful when merging in a number of different Visual FoxPro reports to build an executive package. You can also merge in PDFs generated from other applications like Word, Excel, or other custom Visual FoxPro applications. The source of the PDF files or

the method used to create the PDF does not matter. One example of this could be a header page template generated from Word with some nice graphics and some fancy fonts. Merge in an introductory letter created in Word and saved to a PDF. The next few pages could be a Visual FoxPro report that outlines sales figures for the region. Merge in some nice graphs that were generated via Automation from the Visual FoxPro custom application to Excel and printed to a PDF. The last merge could be another summary from the Sales Manager created in Word and saved to a PDF file. There is no limitation to the merging other than file size and the amount of disk space.

If you want to merge in a number of PDF files in a directory, you can use code that calls the PDFMerger program. Here is a partial listing of PDFDirectoryMerger.prg:

```
DIMENSION laPDFFiles[1]

lcFileSkeleton = ADDBS(ALLTRIM(tcDirectory)) + "*.pdf"
lnPDFCount     = ADIR(laPDFFiles, lcFileSkeleton)

DO CASE
  CASE lnPDFCount > 1
    lcLastFile = tcDirectory + laPDFFiles[1, 1]

    FOR lnCount = 2 TO lnPDFCount
      IF lnCount = lnPDFCount
        * Last one, used the specified combine file
        lcCombinedFile = tcPDFCombinedFile
      ELSE
        * Build a temporary
        lcCombinedFile = FORCEEXT(ADDBS(SYS(2023)) + "Temp" + ;
                                  ALLTRIM(STR(lnCount)), "PDF")
      ENDIF

      lcResult   = PdfMerger(lcLastFile, ;
                             tcDirectory + laPDFFiles[lnCount, 1], ;
                             lcCombinedFile)
      lcLastFile = lcCombinedFile
    ENDFOR

  CASE lnPDFCount = 1
    COPY FILE laPDFFiles[1, 1] TO tcPDFCombinedFile

  OTHERWISE
    * Nothing to do with no files in directory
ENDCASE
```

The program loops through all the PDF files in the specified directory and merges them into one file (based on a parameter passed to the program).

Conclusion

This chapter demonstrates a number of ways to integrate Adobe Acrobat technology with custom Visual FoxPro applications. The ideas presented show alternative methods of generating reports, emailing report output, displaying reports in preview mode without the Visual FoxPro report preview limitations, and capturing information from the users and presenting the same information using Acrobat Forms. We hope you enjoyed reading it and

that you have some idea to how to integrate the power of Acrobat PDF technology with your custom Visual FoxPro applications