l e x a n a \ n e t

June 2005

Custom Properties in Office Documents

By Drew Hamre

[Note: This paper's focus is chiefly on legacy file formats for Microsoft Office 1997-2003.]

email

It's now common practice to pre-process documents before they're loaded into a document manager. Corporations add custom metadata to support versioning, to track provenance, to improve search behavior (e.g., tagging each document with category and topic descriptions), or to integrate with custom editing tools.

The default file formats used in Microsoft Office encourage this practice, because a limitless number of user-defined, custom properties can be added to a file without otherwise affecting a document's appearance or behavior. The same extensible architecture (*structured storage files)* that supports custom properties also allows Office users to embed arbitrary objects within documents (as when, for example, a spreadsheet is stored within a Word document). These structured storage files are commonly called *compound documents*.

This paper will discuss *bulk* pre-processing techniques for Office documents, presenting VB6 and VB.NET software to load custom properties for each document within a staging folder. We'll also discuss how these properties are made accessible via common platforms like Exchange.

We'll also step back and review the *structured storage file* (SSF) architecture, which is used in various Microsoft Office and MSI installer file formats, and in products from Adobe and other vendors. We'll review SSF's history, its programmatic interfaces, and several tools that explore these arcane data containers. We'll also preview changes to the default file formats in Office12 (the next release), which will transition from SSF to compressed XML.

Defining and loading properties for all documents within a folder

From an operational perspective, Microsoft distinguishes four types of Office properties: *automatically updated properties* (such as file size and date), *preset properties* (such as Author and Title), *custom (user-defined) properties*, and *document library properties* (which are exposed in document manager views). These classifications are misleading, however, as they confuse properties maintained by the file system (or document manager) with properties maintained by Office. Below, we'll concern ourselves solely with document properties managed by Office applications and stored within the document file itself, including pre-defined properties (e.g. Author, Title) and user-defined, custom properties.

These properties can be manipulated from within Office applications, or via Explorer shell extensions (right-click | *Properties* | *Summary* or *Custom*). These tools are sufficient to manage properties on a file-by-file basis, but not when pre-processing documents *en masse*. Bulk loads of document stores (where tens or perhaps hundreds of files are copied into a repository in a single operation) are common during the installation of a document management system, or during conversions. During bulk loads, manual operations that are tractable for single files will need to be automated. This includes the act of defining custom properties.

Bulk-loading document properties with either COM/Office 2000 or .NET/Office 2003

The sample 'document pre-processor' discussed below is written as a desktop application. It would be manually invoked by a document library's administrator to iterate through all files in a selected directory (a pre-production staging area), and then create and populate custom properties for each Word document it encounters.

(We won't include code that traverses directory trees and enumerates files, but check here for COM-vintage Visual Basic samples; check here for .NET code samples in both VB.NET and C#.)

In the following, we'll presume that either Microsoft Office 2000, Office XP, or Office 2003 is installed on our document pre-processor system. (Office is required, because our sample uses Word's automation library to manipulate properties.) Despite version differences, Office's object model is remarkably consistent and the code for adding custom properties is virtually unchanged across releases.

Below we'll show a VB6 example of reading/writing Word properties. (For a .NET variant, see here). Both VB6 and .NET versions of the program require that we add a COM reference to '*Microsoft Word Object x.0 Library*' (where x.0 is version 9.0 through 11.0). While COM references from VB6 are straightforward, they are less so from .NET. In this regard, note that Office 2003 is packaged for.NET interoperability, but Office XP requires a separate download here.

The following code shows the key object references we'll need: *Word.Application*, *WordBasic*, and a Word *Document* object.

```
Dim mWord As Word.Application
Dim mWB As Object
Dim objDoc As Document, objLogDoc As Document
Dim CDP As Object
...
Set mWord = New Word.Application
Set mWB = mWord.WordBasic
```

Next, we'll enter a loop that walks a string array filled with target filenames (with full path). For each file in the array, we'll use WordBasic to open the file, make this document active, and then get a reference to its *.CustomDocumentProperties*.

```
For i = 1 To UBound(sFileArray)
   ' ... Open next document
   mWB.FileOpen (m_FileArray(i))
   mWord.Documents(1).Activate
   Set objDoc = mWord.ActiveDocument
   Set CDP = objDoc.CustomDocumentProperties
```

Next we'll test whether a *CustomDocumentProperty* named "XX01" has been defined.

```
   ' ... Check for prior existence of props
   On Error Resume Next
   sXX01 = CDP("XX01").Value
   If (Err.Number <> 0) Then bXX01 = True
   Err.Clear
```

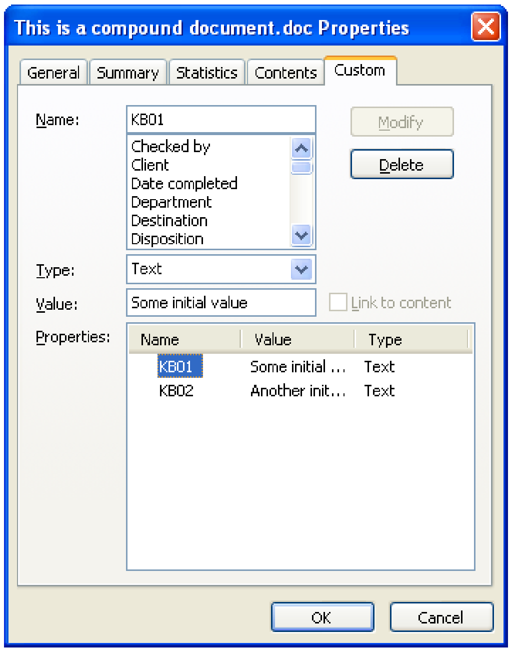
If 'XX01' isn't defined, then we'll do so via the *CustomDocumentProperties.Add* method (which also allows us to set the type to 'string' and load an initial value). Note that in addition to *msoPropertyTypeString*, the types *msoPropertyTypeNumber* (integer), *msoPropertyTypeFloat* (real), *msoPropertyTypeBoolean* (Boolean), and *msoPropertyTypeDate* (date/time) are also supported.

```
   ' ... Add properties if needed
   If (bXX01) Then
      CDP.Add Name:="XX01", LinkToContent:=False, _
         Value:=sXX01, Type:=msoPropertyTypeString
      objDoc.Saved = False
   End If
```

Amended and mended

We should note that three elements in the above code were added as workarounds for problems encountered when automating Microsoft Word:

- We used WordBasic to open the document (rather than Word itself) to circumvent a Word automation problem that mishandles document 'Read-Only' recommendations.
- We explicitly marked the document as 'dirty' (*objDoc.Saved = False*) to avoid a problem in Word 97, 2000, and 2002 that wrongly ignored property changes when determining file-save necessity.
- We tested each Custom Property's existence before creation it so as to avoid errors that would occur if creating a pre-existing property.



Name: KB01

Checked by
Client
Date completed
Department
Destination
Disposition

Type: Text

Value: Some initial value

Properties:

Name	Value	Type
KB01	Some initial ...	Text
KB02	Another init...	Text

Figure 1 - Custom Word properties following bulk load

While these workarounds generally target Word 2000 environments, they don't impede work against more recent versions. The above VB6 code was written against Word 2000's library, but it also works when linked to Word 2003's library and when run against documents created by Word 2000, Word XP, and Word 2003. 'Figure 1' displays these properties in a Word 2003 document.

Despite its robustness, the tactic of automating Office isn't always desirable, however: the performance impact is substantial, there are licensing implications, and multi-user, server-side processing isn't robust. Luckily, there's a less intrusive means of accessing Office properties: *DSOFile*.

Manipulating Office properties *without* launching Office: Using DSOFile

DSOFile is a Microsoft utility that reads and writes both standard and custom Office properties. DSOFile is packaged as an 'automation-friendly' ActiveX DLL, that's readily callable from both COM-vintage code and .NET.

DSOFile is available via free download, and the install package includes a) C++ source code for the DLL, b) sample VB6 client code that calls DSOFile, and c) sample VB.NET client code that calls DSOFile. As the examples make clear, DSOFile provides a clean interface for manipulating custom properties (the following is VB6):

```
Private m_oDocumentProps As DSOFile.OleDocumentProperties
...
' Add the property...
Set oCustProp = m_oDocumentProps.CustomProperties.Add(sName, vValue)
```

Of course, the real complexity of DSOFile lies in its custom C++ code that parses structured storage files. We'll now review the history and architecture of these objects.

Structured Storage Files: '*Storages*' and '*Streams*'

During Microsoft's Professional Developers Conference in 1993, several 'Cairo' technologies were demonstrated, including an object file system (OFS). Microsoft's OFS never shipped, and the release of its successor (WinFS) is now delayed until sometime after Microsoft's Longhorn is released. (In fact, key observers don't expect WinFS before 2008.)

For users, a key benefit of an OFS is the ability to add extensible metadata to any object, and then to use this metadata seamlessly in managing files. Some of these benefits are attainable by using SSFs, as we've seen, and it's probably not an accident that SSF's architecture was defined in 1993 as Cairo development peaked.

OLE2 structured storage files are 'file systems within a file'. A single SSF (such as a Word document) can contain many '*storages'* (containers that behave like file system directories); these storages can contain '*streams'* (which behave like files) or additional 'sub-storages' (which behave like sub-directories).

These two SSF objects – storages and streams – can be nested in hierarchies of indeterminate depth, and are at the core of OLE2's ability to link *and embed*. (With embedded documents, a file created by one application (say, a text file from Notepad) is physically stored and visually presented within a different SSF-aware application's file and interface. When the embedded object is activated, its native application is launched.)

An Office compound document contains a root storage object with at least one stream (its native data) along with one or more sub-storage objects corresponding to linked and embedded objects. Each of the embedded objects also is represented by a storage object containing one or more stream objects, and perhaps also containing one or more storage objects. For Office documents, Microsoft also defines two streams for properties: "*SummaryInformation*" and "*DocumentSummaryInformation*".

To access SSFs, Microsoft provides an API as part of the Platform SDK (the OLE2 libraries). This API is not readily callable except from C or C++; other languages (such as Visual Basic or C#) commonly call COM-friendly wrappings of this API, such as provided by DSOFile. On *non*-Windows platforms, SSFs are made accessible via a variety of third-party tools[1].

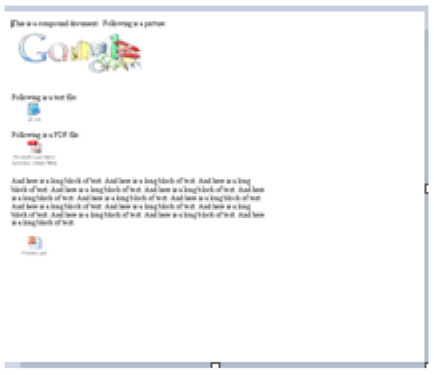
SSF lets multiple data streams 'hide' within what appears to be a single file, and this lack of transparency is controversial (as it was with another little-discussed Windows feature: Alternate Data Streams[2]). The Windows user interface gives no clue when an SSF is encountered, nor whether the SSF contains embedded objects, nor what these objects might be. If a user embeds, say, a spreadsheet within an SSF, and then deletes the standalone copy, there is no mechanism to aid the user in finding and retrieving the file. 'Search' won't find the file, and Explorer provides no hints about where to look.

(SSF's complexity and opacity has raised security concerns. In February, Microsoft released a Security Bulletin concerning a privilege escalation attack that exploited a memory-handling bug for SSFs. Additional security concerns have been raised concerning difficulties in anti-virus 'fingerprinting' for compound documents.)

Although they're not widely distributed, Microsoft offers a pair of utilities (DFView.exe and stg.exe) that can scan SSFs and display their internal storage/stream hierarchy. Ironically, the filename of the lesser known of the pair (*stg.exe*) was also used by the Lovgate virus.

Scanning an SSF with DFView: Where did I hide that presentation?

Microsoft's "DocFile Viewer" utility (DFView.exe) was included with the Visual Studio 6.0 tools (though not with VS.NET) and with the VC6 samples. DFView can 'unpack' the contents of an SSF, displaying the internal streams and storages in an Explorer-like interface.



This is a compound document. Following is a picture.

Following is a text file:

Following is a PDF file:

And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text. And here is a long block of text.

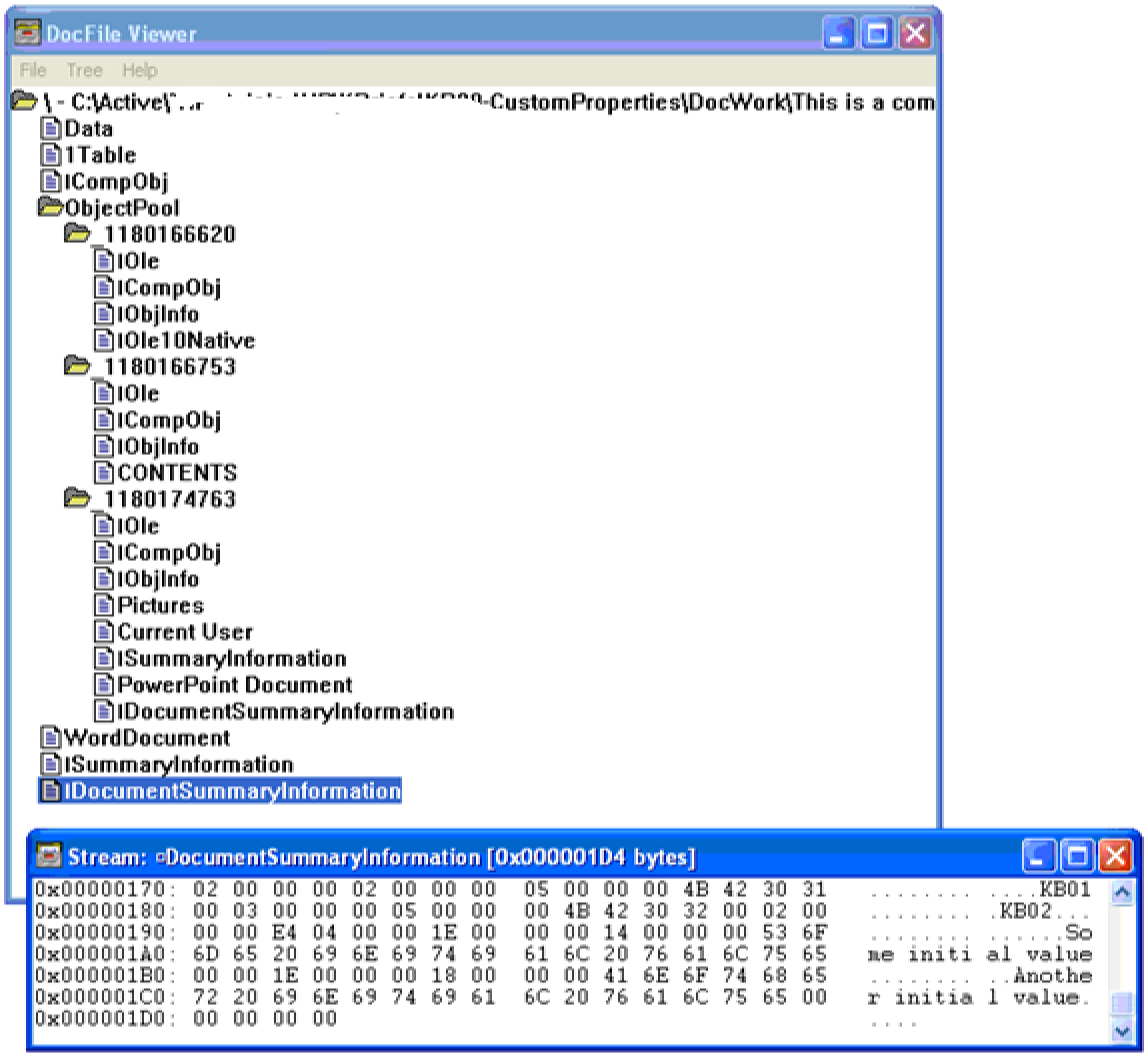
Figure 2 - Word document with embedded objects

Let's use DFView.exe to scan a Word document (see the sidebar illustration) that contains several embedded objects (a bitmap, text file, PDF, and PPT) along with various pre-defined and custom properties.

If we explore this test document's internal SSF hierarchy using DFView, we'll see a visual representation of how the SSF architecture arranges embedded objects. However, we'll see that even DFView does not dispel the mystery surrounding objects within an SSF.

[1]To provide SSF access from pure .NET environments, Desaware offers its StorageTools package, including optional source code. For non-Microsoft environments, the desire to inter-operate with Office documents has motivated a broad range of third-party SSF library development including ports to Java (see here and here), Linux, Gnome, and Tcl.

[2] We should note that SSFs are supported on all Microsoft file systems (i.e., FAT, FAT32, and NTFS), whereas Alternate Data Streams are supported only under NTFS.



```
\ - C:\Active\[illegible]-CustomProperties\DocWork\This is a com
    Data
    1Table
    ICompObj
    ObjectPool
        _1180166620
            IOle
            ICompObj
            IObjInfo
            IOle10Native
        _1180166753
            IOle
            ICompObj
            IObjInfo
            CONTENTS
        _1180174763
            IOle
            ICompObj
            IObjInfo
            Pictures
            Current User
            ISummaryInformation
            PowerPoint Document
            IDocumentSummaryInformation
    WordDocument
    ISummaryInformation
    IDocumentSummaryInformation
```

```
Stream: ▫DocumentSummaryInformation [0x000001D4 bytes]
0x00000170: 02 00 00 00 02 00 00 00  05 00 00 00 4B 42 30 31   ........ ....KB01
0x00000180: 00 03 00 00 00 05 00 00  00 4B 42 30 32 00 02 00   ........ .KB02...
0x00000190: 00 00 E4 04 00 00 1E 00  00 00 14 00 00 00 53 6F   ........ ......So
0x000001A0: 6D 65 20 69 6E 69 74 69  61 6C 20 76 61 6C 75 65   me initi al value
0x000001B0: 00 00 1E 00 00 00 18 00  00 00 41 6E 6F 74 68 65   ........ ..Anothe
0x000001C0: 72 20 69 6E 69 74 69 61  6C 20 76 61 6C 75 65 00   r initia l value.
0x000001D0: 00 00 00 00                                        ....
```

Figure 3 - DFView's display of a Word document with embedded objects and custom properties.

In the above screen-shot, we've exposed our test document's SSF storage hierarchy by opening the DFView folder tree. Here we can see that the embedded objects (text file, PDF, and PPT) have been persisted as 'sub-storages' within a storage called *ObjectPool*. The document's graphic (a device independent bitmap) is not treated as an embedded object, but rather is part of the file's native data stream.

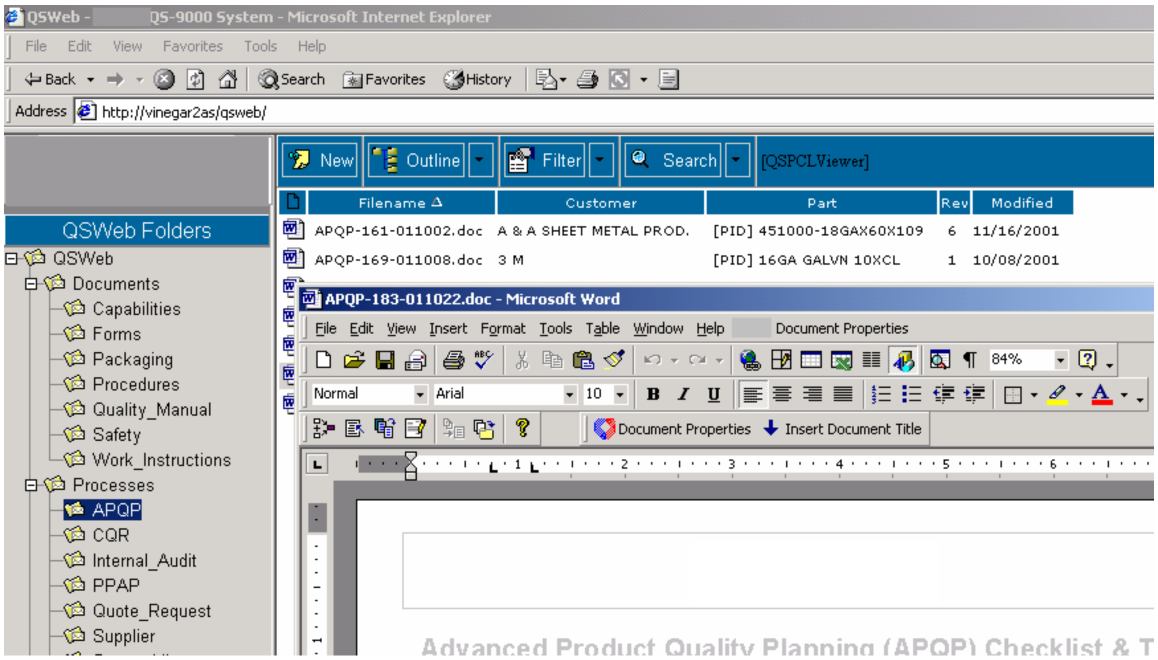
The custom properties we defined and loaded (via the VB6 utility shown earlier) are readily apparent, as they're stored in name-value list format within a special stream called *DocumentSummaryInformation*.

We should also note what *DFView* does *not* show: it doesn't display the names of embedded objects, it doesn't show the files from which they were loaded, nor the date they were added to the SSF. In other words, if we attempt to use DFView to answer our question, "Where did I hide my presentation?" we may find, sadly, that it's not much help at all.

‘Property Hoisting’: Exchange and Office extensions

Custom document properties (unlike the less-identifiable embedded objects) can be surfaced reliably in application interfaces. (‘Property hoisting’ was once the phrase that described this process, though it’s apparently fallen out of favor.) Microsoft Exchange Server’s message/object store has long supported such applications.

A custom Exchange 2000-based custom document manager can use Public Folder views to display both standard and custom properties for stored material, and allow users to sort, select, and report on this information.


QSWeb - QS-9000 System - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://vinegar2as/qsweb/
New Outline Filter Search [QSPCLViewer]
QSWeb Folders
QSWeb
Documents
Capabilities
Forms
Packaging
Procedures
Quality_Manual
Safety
Work_Instructions
Processes
APQP
CQR
Internal_Audit
PPAP
Quote_Request
Supplier
APQP-183-011022.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help Document Properties
Normal Arial 10
Document Properties Insert Document Title
Advanced Product Quality Planning (APQP) Checklist & T

Filename	Customer	Part	Rev	Modified
APQP-161-011002.doc	A & A SHEET METAL PROD.	[PID] 451000-18GAX60X109	6	11/16/2001
APQP-169-011008.doc	3 M	[PID] 16GA GALVN 10XCL	1	10/08/2001

Figure 4 - Custom document manager based upon Exchange 2000 Public Folders with ‘hoisted’ custom properties. Word has been extended with a custom toolbar that interoperates with these custom properties.

The desktop interface to this repository (above) features an Office extension implemented as a custom toolbar. This extension helped guide editing behavior based upon the value of custom document properties, such as document approval status, review distributions, and so on.

New direction for Office12: Default format will be ZIP’d XML, not SSF

Microsoft has twice made highly publicized attempts to store Office documents in formats other than SSF. Word 2000 documents could be saved (with fairly high fidelity) as HTML. Nowadays, Microsoft’s focus is XML, and most Office 2003 applications vigorously support this format.

The trend toward XML storage will solidify in Office’s next release (‘Office12’). According to Microsoft, the default native file format in Office12 for Word, Excel, and PowerPoint is changing from binary SSF to compressed XML, a change that observers believe is the most far reaching design decision of the Office12 release.

Microsoft characterizes the new file design thusly:

- ZIP container with compression
- Multiple XML parts describing file data, metadata, customer data
- Non-XML parts supported as native files (images, OLE objects)
- Relationships define file structure

Microsoft touts the new format as being 'tightly integrated but modular and highly flexible', in addition to being more interoperable, robust, efficient, and secure. It seems the increased attention to intelligible metadata should improve manageability of embedded objects, so perhaps we'll find that hidden presentation after all.

Summary

We've reviewed Office custom document properties and techniques for bulk-loading these attributes. We've reviewed *structured storage files* (the arcane architecture that supports Office custom properties and object embedding) and we've previewed Microsoft's plans to use zipped XML storage as the default in Office's next release.

Additional Background

Mackenzie, Duncan et al. Word 2000 VBA Programmer's Reference. Birmingham, UK: Wrox Press, Ltd., 2001.

Reading and Writing Custom Document Properties in Microsoft Office Word 2003 with Microsoft Visual Basic .NET by Frank Rice
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odc_wd2003_ta/html/odc_wdcustprop.asp

Understanding Custom Document Properties in Microsoft Office Word 2003 by Frank Rice
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odc_wd2003_ta/html/odc_wdcustprops.asp

Acknowledgement Thanks to reviewers for their time and suggestions; any inaccuracies remain mine.

Drew Hamre is a principal of lexana\net and lives in Golden Valley, Minnesota.