StorageTools™

Version 3.0

for Visual Basic

by

Desaware, Inc.

Rev: 3.0.1 (06/2005)

Desaware, Inc.
Software License

Please read this agreement. If you do not agree to the terms of this license, promptly return the disk package and all accompanying items to the place from which you obtained them for a full refund.

This software is protected by United States copyright laws and international treaty provisions.

This program is licensed to you for your use. If you, personally, have more than one computer, you may install it on all your computers as long as there is no possibility of it being used at one location or on one computer while it is being used at another. Just as a book cannot be read by two different people in two different places at the same time, neither can this software be used by two different people in two different places at the same time. You may (and should) make archival copies of the software for backup purposes.

You may transfer this software and license as long as you include this license, manual, the software and all other materials and retain no copies, and the recipient agrees to the terms of this agreement.

You may not make copies of this software for other people or make copies of the manual. Companies or schools interested in multiple copy licenses or site licenses should contact Desaware directly at (408) 404-4760.

If you use this product to create a compiled Visual Basic program that you will distribute as an executable (.exe) file, you may include with your program a copy of the dwReg.dll, dwReg16.ocx, dwReg32.ocx, dwStg16.ocx, dwStg32.ocx, dwStg.dll or dwProp.dll (the runtime libraries). In this case you must include a valid copyright notice on all copies of your program. This can be either your own copyright notice, or the copyright notice that is on each StorageTools medium. You may not modify the dwReg.dll, dwReg16.ocx, dwReg32.ocx, dwStg16.ocx, dwStg32.ocx, dwStg.dll or dwProp.dll in any way.

Limited Warranty

Desaware, Inc. warrants the physical medium and documentation enclosed herein to be free of defects in materials and workmanship for a period of sixty days from the purchase date.

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective diskette(s) or documentation and shall not include or extend to any claim for or right to recover any other damages, including but not limited to, loss of profit, data or use of the software, or special, incidental or consequential damages or other similar claims, even if Desaware, Inc. has been specifically advised of the possibility of such damages. In no event will Desaware, Inc. 's liability for any damages to you or any other person ever exceed the suggested list price or actual price paid for the license to use the software, regardless of any form of the claim.

DESAWARE, INC. SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Specifically, Desaware, Inc. makes no representation or warranty that the software is fit for any particular purpose and any implied warranty of MERCHANTABILITY is limited to the sixty-day duration of the Limited Warranty covering the physical diskettes and documentation only (not the software) and is otherwise expressly and specifically disclaimed.

This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

This License and Limited Warranty shall be construed, interpreted and governed by the laws of the State of California, and any action hereunder shall be brought only in California. If any provision is found void, invalid or unenforceable it will not affect the validity of the balance of this License and Limited Warranty which shall remain valid and enforceable according to its terms.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable. Contractor/Manufacturer is Desaware, Inc., 3510 Charter Park Drive, Suite 48, San Jose, CA 95136.

Table of Contents

Customer Support

We at Desaware have one very simple company policy - we do our best to treat our customers as we would like to be treated (after all, we are programmers too).

StorageTools is a sophisticated product, and while we have checked it in as many environments as possible, it is conceivable that some bugs slipped through our testing process. Once we are able to duplicate a problem, we will provide you with a fix as quickly as possible.

If you have purchased this software directly from Desaware and you feel that StorageTools is not for you or you are otherwise dissatisfied, please feel free to return it for a full refund (if you purchased it elsewhere you will need to contact your dealer for return or refund information - also, we reserve the right to limit this offer to 30 days from the invoice date). Your satisfaction is important to us, and we are well aware that this is a very unusual product and not appropriate for everyone.

Register! Register! Register!

We've found that the person who ends up using a software package is frequently not the person who bought it. Therefore we really need your registration card. This will allow us to send you information about upgrades.

But we can't send this information to you without knowing who you are!

And please send us your suggestions for features that you would like to see in future editions of this project!

Installation

Please refer to the readme file on the CD for the latest information regarding installation.

- Place your StorageTools CD in your drive.
- Use the Windows Start Menu-Run command to run setup.exe from the drive being used. You can also use Explorer to run this program.
- Select the desired setup option to start the installation.
- The setup program will prompt you for a destination directory for the StorageTools files. The default directory is Stgtools.
- Follow any further directions in the setup program. A summary of files installed will appear in file install.log in your StorageTools directory.
- Installation programs are tricky - and we have found that occasionally a system is configured in such a way that the installation program fails. Please refer to the readme file for the latest information on these situations, and for instructions for manual installation.
- The directory containing the StorageTools sample files may contain files **readme.txt** or **readme.wri**. These files, if present, will contain recent information that could not be incorporated into the manual at time of printing. Use the Windows 'Notepad' program to view readme.txt, and the Windows 'WordPad' program to view readme.wri.

File Descriptions and Redistribution Terms

The following files and controls may be distributed with your compiled Visual Basic application without payment of license fees according to the terms on the License Agreement.

dwStg.dll	Component version of the Storage control. It is usually installed in the System directory. It will be registered in the Registry by the installation program.
dwStg32.ocx, dwStg16.ocx	ActiveX versions of the Storage control. These are usually installed in the system directory. They will be registered in the Registry by your installation program.
dwProp.dll	File Property component. It is usually installed in the System directory. It will be registered in the Registry by your installation program.
dwReg.dll	The Registry component. It is usually installed in the System directory. It will be registered in the Registry by your installation program.
dwReg32.ocx, dwReg16.ocx	The Registry control. These are usually installed in the system directory. They will be registered in the Registry by your installation program.
dwAddr32.dll, dwAddr16.dll	These libraries used in programs that read and write User Defined Types to Structured Storage files, and for memory access. They are not used otherwise, and do not have to be distributed with the Storage control unless the functions in them are directly accessed.
OLE32.dll	This file contains the core of OLE. If you are distributing the component version of the Storage control (dwStg.dll), then you may have to distribute a recent version of the file.
MFC40.dll (32 bit), MSVCRT40.dll (32 bit), OC25.dll (16 bit)	These files are required for our ActiveX controls. They will also need to be distributed with your application if you use the ActiveX controls.

The following files may NOT be distributed and are subject to the terms in your license agreement. However, you may incorporate the source code from these files (where applicable) into your own programs.

StgDes32.dll, StgDes16.dll	Design time license file for the Storage and Registry controls.
PropDes.dll	Design time license file for the Property component.
RegBrwsr.vbp and associated files	Sample code for a full-featured Registry editor.
StgBrwsr.vbproj and associated files, StgBrwsr.csproj and associated files	Sample code for a program that views the internal structure of Structured Storage files written in the .NET languages VB.NET and C#.
StgBrwsr.vbp and associated files	Sample code for a program that views the internal structure of Structured Storage files.
PropBrowser.vbproj and associated files, PropertyBrowser.csproj and associated files	Sample code for a program that searches for files on the computer with particular file properties written in the .NET languages VB.NET and C#.
PropBrowser.vbp and associated files	Sample code for a program that searches for files on the computer with particular file properties.
OleObjct.vbp and associated files	Sample code showing how to save objects in OLE Container Controls into a single Structured Storage file.
Records.vbp and associated files	Sample code showing how to read and write User Defined Types to and from a Structured Storage file.
ClsTst.vbp and associated files	Sample code of a Visual Basic class that reads and writes information to a single stream. This allows you to write many different classes, each of which handles its own file access while only requiring one file.
dwProp.hlp	On-line help file for the Property component
Storage.hlp	On-line help file for StorageTools. This file contains context sensitive help for the Storage control, Registry Control, Storage Browser and Registry Browser.

Distribution of Applications That Use StorageTools Components

When you ship programs that use our components, you have to make sure that the StorageTools components are properly installed on the user's machine.

First, the control must be registered. This is usually done by the installation program you use. If you need to register a component manually, you can use the command line program, regsvr.exe (for 16 bit components) or regsvr32.exe (for 32 bit components). These programs can usually be found in the Visual Basic setup kit directory. To use them, bring up a DOS prompt in the same directory as the component and the regsvr program. Run regsvr with the name of the component as the command line parameter. You can also unregister components by using the '-u' flag before the name of the component. In either case, you will get a message box telling you if the action was successful or not.

Second, you must ensure that the files upon which our controls are dependent are also installed on the machine. For the 16 bit controls, the only file that is required is the OC25.DLL library version 2.54 or later. If you are distributing a program using the 32 bit ActiveX controls, you must also install MFC40.DLL (version 4.0.5277 or later) and MSVCRT40.DLL (version 4.0.5270 or later). Do not distribute version "4.20 - OS USE ONLY DO NOT DISTRIBUTE" of MSVCRT40.DLL. This file is meant for use in Windows NT 4.0 only, and it will take special care to get it to work in Windows 95 or Windows 98 machines. It is usually smart to distribute the lowest version of these files that will work - any good installation program will not replace existing files of later versions, so this technique will cause the least amount of change in the user's machine.

If you are distributing the Storage component (dwStg.dll), the Registry component (dwReg.dll) or the Property component (dwProp.dll), then you may have to ship the OLE32.DLL library. If you use the Summary Information methods (those that begin with "si" or "dsi" in the dStorage object) or the Property component, then you need to make sure the user's machine has version 4.0 or later. Windows NT 4, Windows 98 and Internet Explorer all ship with version 4.0 or later of this library. If you do not use these functions, then you will not get any problems even with older versions of OLE32.DLL.

You can locate the version number of a file by looking at the file's properties in the Windows Explorer program.

Differences Between StorageTools 2.5 and 3.0

1. The components have been modified to work in the .NET environment, including Finalize methods where appropriate.
2. .NET versions of the samples are included.
3. StorageTools has new methods for reading and writing variants holding data larger than MAX_INT number of bytes (32000 bytes). They are VariantPutEx and VariantGetEx.
4. StorageTools has a new method (IStreamToDStream) for converting a pointer to an instance of an OLE IStream interface into a dStorage object.

Persistence of Data - A Technical White Paper

When should your VB programs use Databases, the Registry, .INI files, or Disk files to save information? And what is OLE Structured Storage? Find out in this new article by the author of the Visual Basic Programmer's Guide to the Win32 API.

by Daniel Appleman

==========

The following technology white paper is based on a talk that I presented at the Silicon Valley Visual Basic User's Group in January 1996. It is being distributed in an effort to educate Visual Basic programmers regarding available techniques for implementing data storage in their applications. The paper focuses on matching an appropriate data storage technology to the needs of individual applications, and discusses the advantages and disadvantages of each approach. The end of this article does discuss a Desaware product (the company for which I work), however you will find that the bulk of the article is an objective and technical discussion on storage techniques that should prove valuable to every Visual Basic programmer. By reading this article you will gain:

An understanding of many of the tradeoffs involved in data storage technologies including initialization files, the system registry, disk files, database systems and structured storage systems.

An excellent understanding of OLE structured storage as it relates to application design - including where its use is and is not appropriate. Also, you will see how OLE structured storage fits into Microsoft's OLE strategy.

Introduction

OLE structured storage is one of the most exciting technologies to come along in quite a while. It will dramatically change the way you work with files. It is.....

Wait. Isn't this the way most articles begin when describing a cool technology? They focus on the features and capabilities of the tool or technique being described. But is this really the information that most programmers need?

You see, most programmers (myself included) tend to fall in love with technology. We hear about a cool new tool or technique, and it becomes, for a while, the greatest thing that we've ever heard of. When I was studying computer science I remember being introduced to a new language every week or two, and each one instantly became the "best" language in my repertoire - the language that I could surely use to perform any task more efficiently and quickly than any other. And it would remain the "best" language until the next one came along.

This is something that I think is common among many programmers and engineers. We become passionate about technology. We tend to love our work. Our preferences of languages, tools and machines are almost religious in their intensity (in fact, I dare say that there are programmers who are considerably more passionate about their programming tools than their religion).

So before discussing this "cool new tool" called OLE structured storage, I think it is important that we pause for a moment and place it in perspective.

You see, as programmers our real job is not to fall in love with technology. It is to solve problems using software. And as such, it is our professional responsibility to choose the tools that will best help us solve the problem at hand - not the one that we happen to be excited about at the moment, or the one that the industry is busy hyping.

That said, here is the truth about OLE structured storage: It is a powerful technology that is highly suited to solving particular problems, and totally unsuitable for others. So, how does one determine which problems are right for this technology? Let us start with considering the entire class of problem at hand: that of persistence of data.

Persistence of Data

Imagine a program such as a word processor that was unable to save information from one session to the next. With such a program, you would have to type a document from scratch every time you needed a printout - roughly akin to the way offices used typewriters just a few decades ago. Much of the value of computers comes from the ability of applications to store information - to persist their data. There are many technologies available for persisting data, and not all of them are appropriate for every situation.

1. There are three major reasons for persisting data in an application and one or all may be appropriate for any given program. They are as follows:

2. Application configuration information: Many applications save state information from one session to the next. It may remember user settings such as whether a backup should be generated for each file, or the positions of windows so that the application will restore itself to the same state that it was in when it was last closed. The application might save general configuration information as well as user specific configuration settings (for applications that support multiple users).

3. Documents: Many applications work on documents that are opened at the beginning of a session and saved or closed when the session is over. Typical documents are word processing documents, spreadsheets, pictures and so forth. Documents can often contain many types of objects. The key thing to remember about documents is that they generally correspond to a single disk file. This makes a document easy to manage as an individual unit. For example: it can be copied to a diskette, or sent via Email.

4. Records: Applications that need to keep track of multiple records which are similar to each other will typically use some sort of database storage mechanism. Databases support searching and filtering and are designed to be easily shared by multiple applications simultaneously.

Each of these reasons for storing data influences the choice of technology that can best solve the particular storage problem. But the choice of technology is not always obvious. Let's take a look at the major storage technologies in use under Windows and how appropriate each is for the problems listed above.

Private Initialization Files

Initialization files are an older technology that goes back to Window 2.0 (version 1.0 did not support private initialization files, only win.ini). While some of the newer Windows documentation suggests that one should always use the registry instead of initialization files, this is not necessarily the case. Initialization files maintain a number of advantages over the registry for some applications. They can be edited by hand using any text editor such as Notepad. They can be easily copied or saved using simple file commands. They are very easy to work with using a few simple API functions or Visual Basic commands.

Are initialization files appropriate for saving application configuration information? Yes - absolutely. A program's state can often be stored as a relatively small number of short string entries or numeric entries - exactly the kind of information that initialization files handle best. It would take more effort to handle user specific configuration information - each user would have to have their own section in the file, but it is still a viable solution.

Are initialization files appropriate for saving documents? Probably not. Their performance is not adequate for most documents. Each entry is limited in size, as is the entire file (depending on the operating system). They can only handle text data, limiting their flexibility for general use.

Are initialization files appropriate for saving record information? Certainly not, except perhaps for the most limited applications.

The System Registry

The system registry or registration database provides hierarchical storage in a central location on each system. The registry is quite limited under Windows 3.x, supporting only limited data types and only one value at each node in the hierarchy. Under Windows 95 and Windows NT, however, each key in the registry may have any number of named values associated with it, and those values can be one of a wide selection of available data types including raw binary data. Each value is generally under 2k bytes in length.

Is the registry appropriate for saving application configuration information? Absolutely. It provides a far more sophisticated structure for saving configuration information and you can easily save user specific information by simply giving each user their own node in the registry hierarchy. The registry is, however, harder for an end user to edit (which is either an advantage or disadvantage, depending on your application). Editing the registry always creates the risk that the user might accidentally make a change that could interfere with the correct operation of the system - since the registry, unlike private initialization files, also contains configuration information for the system itself. The registry is harder to access programmatically than private initialization files, requiring the use of more sophisticated API commands (though a simple set of Visual Basic commands are also available for basic registry operations).

Is the registry appropriate for saving documents? Certainly not. There is no clean mechanism for managing individual parts of the registry - how would you copy part of the registry to a disk file or send it through Email? The registry is, by its very nature, not document oriented. It is also strongly recommended that each value in the registry be kept relatively small.

Is the registry appropriate for saving record information? Not a chance - all the reasons discussed for documents apply here as well.

Independent Documents

One of the most common methods used by applications to persist data is to store the required information in document files on disk. There are a number of standard formats for disk files, for example: .BMP files (Windows bitmap format) is a standard file format for bitmap images. RTF (rich text format) is a standard formatted text document format. In many cases, applications define their own private file formats - in fact, many standard formats originated in that manner. Whether a private or public file format is appropriate for a given application is one of the design choices that must be made by the developer.

With document files, the contents of the file are the responsibility of the application, as are all of the file operations. File operations are implemented using either API commands or a wide variety of Visual Basic commands that allow you to work with lines, records or binary data.

Are document files appropriate for saving application configuration information? Yes - after all, a private initialization file is, essentially, a document on disk. But document files are rarely used in this way. If the configuration information is simple, it is easier to use an initialization file. If the configuration information is complex or requires the persistence of data that cannot be easily represented as text, it is often easier to use the system registry.

Document files are easy to manage on disk. They can be copied, moved, sent via Email, and managed using standard file management software.

The task of reading and writing document files can range from very simple to extremely complex, depending on the type of document. A simple text document can be created by simply printing lines of text into the file, each one separated by a carriage-return line-feed combination. If, however, you wish to insert a line into the middle of the file, or remove a line from the file, you will generally need to write the entire file from scratch - there is no easy way to shift information in a disk file.

What if you wish to save many different types of information in the document? For example: a word processing file that contains both text and pictures? In that case you need to define a more complex file format that keeps track of where individual objects are located within the disk file. You might need to actually define an internal directory within the disk file that lists the various parts of the file and where they are located. In fact, if you think about it, you might reach a level of complexity in file organization that is used by database systems (which routinely track large numbers of arbitrarily sized objects within a disk file).

Which brings us to the subject of records. Is a document file appropriate for storing multiple records of data that are similar to each other? The answer here is yes and no. Let's say your application uses a user defined type (data structure) internally to store data, and needs to persist an array of these structures. In this case, a document file can be an extremely efficient way to store this information. Visual Basic includes record I/O file operations that are fast and have low overhead.

The document approach towards storing records does, however, have a number of potential disadvantages. Insertion and deletion of individual records can be complex. If there is a chance that the structure format might change, your application will need to keep track of document versions and implement a scheme for updating documents from one version to the next. Use of variable length records requires a more complex storage scheme that includes some mechanism for finding the start and end location of each record. Also, any searching or filtering schemes need to be implemented by the application.

Records

In many cases you may need to persist groups of records, where the records are similar to each other in structure. These often take the form of tables of records, where each record contains the same fields. You may need to work with different types of records in the same file (multiple tables). In many cases, you will want the information to be accessible to multiple users or applications at once. Clearly, this is the type of information that is often stored using a database system. This might include use of a specialized database command set or the database controls and commands built into Visual Basic.

You may be thinking that this is a somewhat convoluted way of talking about databases. Why go to all of the trouble to differentiate between "saving records" and building a database? Because it is important to distinguish between the goal and the solution. The goal is to persist a certain type of data: records that are similar to each other. One solution is a database - but it is not the only solution.

You have already seen that it is possible to store records in a document file. However, if a database is clearly designed for record storage, why would you ever want to store records in a document file? You might want to do so if you do not need the database's searching and filtering capability, if you have little or no need to share the file and if the records are simple and of fixed length. In that case not only might the document approach be easier to program - it will probably be faster and will allow you to avoid the overhead of including a database engine with your application.

The case for application configuration information is easier to see. A database would almost always be extreme overkill for this type of data persistence. Even storage of per user application information is more easily accomplished using any of the other techniques.

What about documents? Is it appropriate to use a database to store document information - a word processing document, for example? It certainly can be done. You can imagine a complex database structure that includes one table for the lines of the document (perhaps with formatting information), and other tables that contain different types of objects, perhaps in blob format to hold images, video, sound, and so forth. These tables can be linked through the relational capabilities of the database.

This approach would work - but it can become quite complex - easily as complex as implementing the document using a proprietary format and performing your own disk operations. The database approach also can have a significant impact on performance and involves significant overhead.

The Missing Technology

All programmers have to deal with the problem of persisting data, and there is a tendency to jump straight to a solution without considering the problem in the context of the requirements of a particular software project. It is true, as you have seen, that application configuration information is usually best handled using either a private initialization file or the system registry. It is true that record storage is usually best handled by a database. It is true that simple documents are usually best handled by reading and writing a disk file directly. But you have also seen that each of the four technologies that we've discussed can be applied for any of these persistence tasks, and the obvious choice is not always the best.

It is also clear from the previous discussion that there is one type of persistence problem for which none of the technologies is ideal. What is the best way to store complex documents that contain different types of information? Implementing a complex file format is a great deal of work. And databases are designed for records that are similar to each other - using them to store arbitrary types of information of arbitrary length is possible in many cases, but can be even more work.

Until recently, there has been no obvious solution to this problem available to Visual Basic programmers.

Before looking at OLE structured storage in the context of this problem, let's pause for a moment and consider the characteristics of the ideal complex document storage system - a system able to manage a complex document file. Assume for the moment, that the term "object" refers to any block of data that you care to define.

- The file must be able to hold a very large number of objects, of types defined by the application. Each of the objects must be of any arbitrary size, efficiently handled whether it is a few bytes or megabytes in size.

- It must be possible to insert and remove objects, or change their size or contents without needing to rewrite the entire file.

- It must be possible to write data into the space allocated for an object, without risk of interfering with other objects stored in the file.

- The storage system must maintain an internal table of information about the objects in the file, making it easy to find and reference any object. It would be even nicer if it were possible to organize these objects in some hierarchical manner of objects and sub-objects.

The complexity of implementing such a system on your own is substantial - and one of the reasons that it is so tempting to look for alternatives to creating a private file format. At the same time, this description might sound familiar - in fact, a storage system that meets these criteria is present on every Windows and DOS based system.

It is the file system.

Think for a moment of files as objects. A file system can handle many thousands of objects. Each file can range from zero bytes to gigabytes in size. You can certainly add and remove files without fear of one file interfering with the next. And writing into one file cannot corrupt any other. Files are named, and can be organized in a hierarchy of directories.

In fact, you will find some applications that solve the problem of complex document storage by using the file system - dividing their documents into multiple disk files. An example that I worked with recently is Corel Ventura Publisher, in which one publication can consist of multiple chapters each of which contains links to documents of different types. A single publication can thus be made up of hundreds of files, which are typically organized in one or more directories.

The disadvantages of this approach are clear when you try to copy a document to a floppy disk, to another directory, or to send it via Email. Copying the publication requires a special utility that can search out all of the linked files and copy them to the correct destination. If even one file is missing, the entire publication can fail to load at worst, have missing information at best.

Which brings us to OLE Structured Storage.

OLE Structured Storage

It sometimes seems that part of the challenge of being a Windows developer (whether you use Visual Basic or another language) is coping with the multitude of new terms and acronyms that are generated with great frequency from Redmond. OLE Structured Storage is one of these.

Fortunately, this is where the somewhat long introductory material that you have read up to now serves its purpose. Because it makes it possible to understand exactly what OLE Structured Storage is and why it exists and even how it might fit into your own development plans.

Think again, for a moment, about a file system - both its advantages and disadvantages, and how it might be used to address the problem of creating complex documents. Consider then, that a file system exists on a hard disk drive - a large area that is able to store information. What would happen if, instead, you could place an entire file system within a single disk file? Suddenly you would have all of the advantages of a file system, with none of the disadvantages. Since the entire file system is within a single disk file, you can copy, move and Email all of the objects for that file system by simply copying, moving or Emailing the file that contains the file system.

OLE Structure Storage is a technology that places an entire file system within a single disk file. Nothing more and nothing less. And understanding that this is what OLE structured storage is, you know nearly as much about it as any of the experts.

Terminology

In a file system you have directories and files. Each directory has a top level root directory. Each directory can contain multiple subdirectories and any number of files.

In OLE Structured Storage, the term *Storage* is used in place of directory, and the term *Stream* is used in place of file. When working with OLE Structured Storage Files, you start with a top level storage (*root storage)*. This storage can contain additional sub storages and streams. Streams can be created, opened and closed just like files. You can write into streams without worrying that one might interfere with another. OLE Structured Storage takes care of allocating space for streams and storages, and managing space allocation within the disk file.

OLE Structured Storage also defines a name for a disk file that is managed by this technology. These files are called *Compound Documents*.

Why is Structured Storage a part of OLE?

Structured Storage accurately describes what this compound document technology does, but why is this part of OLE? - the "Object Linking and Embedding" standard from Microsoft?

Part of this is the fact that OLE is in fact a collection of technologies - a sort of grab back under which many of the new developments for Windows falls. But part of it has to do with one of the major goals of OLE - the ability for users to deal with documents instead of applications.

Consider what happens when you embed an Excel spreadsheet into a Word document. When you click on the spreadsheet part of the document, the Excel user interface appears within Word - a technology called "in-place editing". The document is a word document, yet clearly, Word is able to somehow store and manipulate the Excel spreadsheet object that is somehow embedded into the Word file. Or can it?

In fact, Word has no idea what data is contained in the Excel spreadsheet or how to work with it. What actually happens is that the Word file is a compound document. One of the storages in the document is a place where objects foreign to Word are stored including, in this case, the Excel spreadsheet. Each object is stored in its own stream. When you save the word document, Word does not know how to store the spreadsheet object, but it does not need to - all it has to do is pass a handle to the stream to Excel. Excel then saves the spreadsheet data into the stream.

When you reload the document, Word looks at the objects in its object pool. Each of the streams starts with a GUID - a universal identifier. Word looks at that identifier and calls the OLE libraries asking it to create an object based on that stream information. OLE looks through the system registry, sees that it is an Excel spreadsheet object, then loads those parts of Excel that are necessary to display the object and allow the user to edit it.

As you can see, OLE Structured storage fills the critical need of allowing an application to store objects even though it does not know their internal format - it can simply pass a stream handle to a server that knows how to work with that object. You can use this same technique in Visual Basic in conjunction with the OLE container control to store foreign objects inside your own compound document files.

A quick look with a compound document browser shows that OLE Structured Storage is used by many Microsoft Applications including most of the Microsoft Office applications.

Should you use a Compound Document or a Database?

I've been asked this question any number of times. In fact, a large part of the purpose of this article is to dispel this question. Note that I say dispel - not answer. The question can't be answered because it is intimately tied to the design of each individual application.

If your application is storing multiple groups of records, where all of the records in a group are similar to each other, and requires sophisticated search and filtering capabilities and support for the definition of relationships between records in different groups (tables), then you probably need a database.

If you wish to create a private document format that contains arbitrary types of data of different sizes, and perhaps objects from other applications, Structured Storage might be a better choice.

In either case, the problem is first to determine the needs of your particular application. Once the requirements are defined, the choice of technology will, in most cases, be clear.

Private vs. Public Information

While the contents of any particular stream in a compound document is defined entirely by the application or object that it contains, the management of streams and storages within the document is the responsibility of OLE itself. This means that it is possible for any application to browse through the structure and contents of the document - just as any application can open and read a disk file.

One of the nice results of this characteristic of compound document files is that you can mix private and public data within a file. You can, for example, place a plain text description in a stream with a given name that any application can read.

Microsoft has actually defined a standard properties format stream called the "SummaryInformation" field. This stream contains standard summary information such as the document Title, Author, Comments and so forth. This means that any application can read this summary information. You can use it in your own documents, or use it to obtain information about other documents that use this feature.

Self Persisting Classes

You've seen how OLE Structured Storage allows each application to be responsible for its own objects within a single document. You can use the same technique within your own application by creating self-persisting classes.

Say you create an application that contains a number of different classes and you wish to persist collections of objects from each class. One technique involves creating high level routines for saving and loading all of the objects. The disadvantage of this approach is that any changes to a class require that you modify the high level routines. This includes adding new classes, or changing the formats that a class uses to save its data. This kind of centralized approach is necessary in most cases where you are creating a private file format that you are managing yourself.

OLE structured storage makes possible a better approach in which each class knows how to transfer its data to and from a stream. In this case, all the high level routine has to do is call a "Save" or "Load" method for each instance of each class. If a class needs to change its data format, it can do so at will (assuming it maintains the ability to load the older format). The high level persistence routines do not need to be modified in any way.

Use of self-persisting classes can improve the long term reliability of an application by making it less likely that the persistence operation of one class might interfere with those of others used by the application - even when the code or data formats for individual classes are modified.

Other Capabilities of OLE Structured Storage

Compound document files support sharing - individual storages can be opened for exclusive or shared access, in much the same way as disk files support sharing.

OLE Structured Storage also supports transactioning. You can open a storage in transacted mode. After modifying the contents of the storage you then have the choice of committing those changes or reverting to the original contents. This can be ideal for implementing features such as incremental Undo operations.

OLE Structured Storage and Visual Basic

There is just one catch that exists when it comes to using OLE Structured Storage technology from Visual Basic. That is that Visual Basic does not support this technology. OLE Structured Storage is not implemented using API calls but rather OLE interfaces, a system not supported in VB.

Fortunately, you can access all of the capabilities of OLE structured storage from Visual Basic using a third party package called "StorageTools" from Desaware. This package includes 16 and 32 bit OLE controls that support both structured storage and easy access to the system registry.

The Need for Structured Storage

Virtually every program needs to read and write information to disk. This means that essentially every programmer has the problem of determining how to organize information within a file. And of course, this file format would, more than likely, be incompatible with that of other programs.

OLE2 Structured Storage is Microsoft's cross-platform standard for organization within a file. This standard offers a number of improvements to normal file storage:

- Organizes information inside files as a directory structure -- a “file system within a file”.
- Supports easy incremental file saves -- allowing you to save only that part of the file that was modified, drastically reducing disk access times.
- Can be written and read from a block of memory, creating a ready made, automatically expandable memory structure that can be easily saved to disk.
- With Transactioning, information put into a structured storage is automatically buffered, and any changes made to it can be undone.
- Because structured storage is a standard file format, it allows any Structured Storage-aware program to find information within a file even if it does not have detailed information about the file format. Different file formats can share a common information block, called a Property Set, allowing you, for example, to look at the title of a word processing document without knowing which program originally created the file. Each application can choose to publish this type of common information if it wishes.

OLE Structured Storage manages the allocation and organization of data within a disk file. This means that if you add data to a stream within a compound document, you need not worry about overwriting other data within the file. The structured storage system will rearrange the contents of the file as necessary.

Compound File Elements and Names

Compound files have two primary structures, "Storages", which act like directories, and "Streams" which act like files. A Storage can contain any number of Streams and Storages.

Each Storage and Stream has a file name that can be up to thirty two characters in length. The filenames can contain any character except "\", "/", ":", "!", "." or "..". The Root Storage's name is the same as the name of the file itself, so it has the same restrictions in length and style as the underlying file system. The first character in a file name can occasionally contain a special value denoting the owner and purpose of that Element. These special values are:

Character	Description
ASCII 1, 2	These elements are for the exclusive use of OLE libraries.
ASCII 3	These elements are for the exclusive use of the OLE Container of the OLE Object that created this Compound File.
ASCII 4	These elements are created by custom Structured Storage implementations.
ASCII 5	This element holds a property set, a method for sharing information about a document.
ASCII 6 - 31	Reserved for future use.

Storages retain the times that they were created, last changed and last accessed (OLE currently does not support the same for Streams). Streams can contain up to 2^{32} bytes each. (That's more than four gigabytes.)

Each open Storage and Stream has its own buffering. Therefore, you must remember to commit all of the parent Storages of an element in order for that element to be saved to disk. For example, if you made changes to a Stream in the root Storage, simply calling the **Flush** method on the Stream is not sufficient to save it to disk -- you must also **Commit** the Root Storage.

Summary Information

Microsoft also created a standardized method of storing information, called "Property Sets", providing a technique for other programs to publish and extract information about a document. Two property sets are common to almost any Structured Storage file. They are called the “Summary Information Property Set” and the “Document Summary Information Property Set”.

The Summary Information property set resides in a single stream named “SummaryInformation”. Most programs expect it to be located in the root storage of a file, but it is not uncommon to see files with multiple SummaryInformation steams. The name, like that of all Property Sets, begins with the special character ASCII 5. In recent versions of Windows, the information in the Summary Information property set is shown when you view the properties of a file.

The Summary Information Property Set contains the following properties:

Property Name	**Type**
Title	String
Subject	String
Author	String
Keywords	String
Comments	String
Filename of template	String
Last author	String
Revision number	String
Total editing time	Date
Date last printed	Date
Date of creation	Date
Date last saved	Date
Number of pages	Long Integer
Number of words	Long Integer
Number of characters	Long Integer
Name of the application that created this file	String
Security Level	Long Integer

Of course, not all of these properties may be useful for a given document. For example, a file dealing with sound effects might not have a word count. Simply leave blank the fields that you do not require.

The Document Summary Information property set was created some time after the original Summary Information set, once Microsoft started using the Structured Storage file system in applications besides Word. It resides in a single stream named "DocumentSummary-Information". Most programs expect it to be located in the root storage of a file, but it is not uncommon to see files with multiple Document Summary Information steams. The name begins with the special character ASCII 5. In recent versions of Windows, the information in the Document Summary Information property set is shown when you view the properties of a file. Unlike the Summary Information stream, the Document Summary Information stream actually contains two blocks of data: a list of attributes similar to the previously described property set and a section for user defined properties.

These are the properties for the Document Summary Information Property Set:

Property Name	**Type**
Category	String
Presentation Target	String
Number of bytes	Long Integer
Number of lines	Long Integer
Number of paragraphs	Long Integer
Number of slides	Long Integer
Number of notes	Long Integer
Number of hidden slides	Long Integer
Number of multimedia clips	Long Integer
Scale image to window or crop it.	Boolean
Manager	String
Company	String
Are links up-to-date	Boolean

User defined properties of the Document Summary Information set can have any name and be a string, date, long integer or Boolean. Some programs such as Microsoft Word provide a ready list of names, but these are no different than any other that name you might use.

The Desaware Storage Control

The Desaware Storage Control (DWSTG.DLL, DWSTG32.OCX and DWSTG16.OCX) provides an easy way to create Compound Documents. It allows you to read compound document files as if they were regular files, using familiar Visual Basic commands. This makes it easy to learn and switch between file formats. The two parts of the Structured Storage specification, "storages" and "streams" are implemented as Visual Basic Objects, allowing the user to create as many storages and streams as are needed (see the "Set" command in the Visual Basic help file for more information). Any number of storages and streams can be kept open, at the cost of only one file handle per Structured Storage file. The control can read and write information within a stream in a number of Visual Basic compatible formats. Document Property Sets can be easily edited. The Storage Control can even place a compound document into a dynamically sized buffer in memory.

Why are There Two Versions of the Control?

We created the original ActiveX version when the ActiveX standard was new. Since that time, there have been a number of changes to Visual Basic and the original standard no longer matches the capabilities of the modern language. It is single threaded, and must be placed on a form in order to work. Its licensing scheme did not allow its use in components. It also requires the programmer to distribute two large Microsoft libraries.

The new component version is designed to provide superior performance while maintaining a similar interface.

How do I Convert an Existing Program That Uses dwStg32.ocx to Use the dwStg.dll Component?

There are only two major changes you must make to your program. First, replace the ActiveX control with the component. Remove any instances of the control from your forms, and uncheck the Desaware Storage control from the Components dialog box in Visual Basic.

Select the "Desaware 3.0 Storage Type Library" from the References dialog box. Where you declare your objects, add a Dim statement for the component. It may make your conversion easier if you use the same name as the control had:

```
Dim dwStorage1 as New dwStorage.
```

Second, you must change the dimension statements for your storage and stream objects. Where they both were declared as "`Object`" under the ActiveX control, now you must specify "`dStorage`" for all storage objects and "`dStream`" for all stream objects.

Those are all the changes you need to make - unless you want to use some of the new features of the component.

What if I Want To Use StorageTools in My Own Ccomponent Which is Then Used in the Visual Basic or .NET Environments?

Like all Desaware components, the dwStg.dll storage component requires that a license file be present in order to work during design time. This poses no problem if you are using dwStg.dll as part of an application or ActiveX server. However, it will cause a problem if you try using it in an ActiveX control, ActiveX DLL or other object that acts as a component. When the end-user attempts to load your component within Visual Basic or the .NET envoronment, the dwStg.dll storage component detects that it is running within Visual Basic and raises a licensing error.

To allow you to use the dwStg component in your own controls, we have created a license scheme that allows you to use it even when a design-time license is not present, just by adding a line of code. See the **EnableComponent** method for more information. Naturally, there is no cost or royalty fee for you to use this mechanism. However, we do remind you that under the terms of the StorageTools license, you may only distribute our components when your component adds significant and primary functionality. In other words, you can't use it to ship your own general purpose Structured Storage access component. This is similar to the license terms that Microsoft and other companies use in cases like this – for example: Microsoft does not allow you to use the JET redistributable components to implement your own general purpose database product. If you have any questions, don't hesitate to contact us.

.NET Storage Component Example

The Storage Component consists of three different types of objects which you can use to access any number of storages and streams. This

simple example illustrates how to create and use these objects. To use the component, first check the item called "Desaware 3.0 Storage Type Library" in the OLE section of the "Add Reference" dialog box. Add the imports line that notifies .NET you are using the StorageTools library:

```
Imports Desaware.StorageTools.dwStg ' VB.NET
using Desaware.StorageTools.dwStg; // C#
```

Next, create a dwStorage object and references for each storage or stream you will need. The component automatically creates the storage and stream objects as needed, so you do not need to use the "New" keyword for them.

```
' VB.NET
Dim dwStorage1 As dwStorage = New dwStorage
Dim RootStorage As dStorage
Dim Stream As dStream

// C#
private dwStorage dwStorage1 = new
   dwStorage();
private dStorage RootStorage;
```

private dStream Stream;Now we will create a file on disk. We will use the Storage Control's method **CreateStorageFile**, which returns a storage object representing the root storage. The first parameter is the filename and the second parameter is the access mode.

```
RootStorage = dwStorage1.CreateStorageFile
   ("c:\test.stg", dwstg.STG_CREATE Or
   dwstg.STG_READWRITE Or dwstg.STG_DIRECT Or
   dwstg.STG_SHARE_EXCLUSIVE) ' VB.NET

RootStorage = dwStorage1.CreateStorageFile
   ("c:\test.stg", dwstg.STG_CREATE |
   dwstg.STG_READWRITE | dwstg.STG_DIRECT |
   dwstg.STG_SHARE_EXCLUSIVE); // C#
```

Now we open a new stream within the storage by using the **CreateStream** function, one of the methods of the Storage Object.

```
Stream = RootStorage.CreateStream
   ("StreamName", dwstg.STG_CREATE Or
   dwstg.STG_DIRECT Or
   dwstg.STG_SHARE_EXCLUSIVE Or
   dwstg.STG_READWRITE) ' VB.NET
```

```
Stream = RootStorage.CreateStream
   ("StreamName", dwstg.STG_CREATE |
   dwstg.STG_DIRECT | dwstg.STG_SHARE_EXCLUSIVE
   | dwstg.STG_READWRITE); // C#
```

The Stream Object's **Put** method is one of the ways to actually place information into the file. However, even in Direct mode, data is still buffered. (See STG_DIRECT in the *Access Flags* section for a discussion of Direct and Transacted modes.) Therefore, after changing the stream, we have to clear the buffers and force the information to disk with the **Flush** and **Commit** methods.

```
' VB.NET
Dim Info as Object
Info = "This string is the information to be
   inserted into the file."
Stream.Put (0, Info)
Info = "This will be added after the first
   sentence."
Stream.Put (dwstg.STG_SEEK_DONTMOVE, Info)
Stream.Flush (dwstg.STG_DEFAULT)
RootStorage.Commit (dwstg.STG_DEFAULT)
```

```
// C#
private object Info;
Info = "This string is the information to be
   inserted into the file.";
Stream.Put (0, Info);
Info = "This will be added after the first
   sentence.";
Stream.Put (dwstg.STG_SEEK_DONTMOVE, Info);
Stream.Flush (dwstg.STG_DEFAULT);
RootStorage.Commit (dwstg.STG_DEFAULT);
```

You have now successfully created a compound document with the file name "c:\test.stg". This compound document has a single stream with the name "StreamName" which contains two single strings.

Finally, always delete Visual Basic objects to free any memory that might be used. An important point is that .NET does not free these objects right away – that takes place the next time there is garbage collection. If you try to use one of these objects before they are collected, it will cause errors. Before you set the objects to nothing, use the **Finalize** method.

```
' VB.NET
```

```
Stream.Finalize()
RootStorage.Finalize()
Stream = Nothing
RootStorage = Nothing

// C#
Stream.Finalize();
RootStorage.Finalize();
Stream = null;
RootStorage = null;
```

Visual Basic Storage Component Example

The Storage Component consists of three different types of objects which you can use to access any number of storages and streams. This simple example illustrates how to create and use these objects. To use the component, first check the item "Desaware 2.0 Storage Type Library" in the References dialog box of Visual Basic.

Create a dwStorage object and references for each storage or stream you will need. The component automatically creates the storage and stream objects as needed, so you do not need to use the "New" keyword for them.

```
Dim dwStorage1 as New dwStorage
Dim RootStorage as dStorage
Dim Stream as dStream
```

Next, create a file on disk. We will use the Storage Control's method **CreateStorageFile**, which returns a storage object representing the root storage. The first parameter is the file name and the second parameter is the access mode.

```
Set RootStorage = dwStorage1.CreateStorageFile
   ("C:\TEST.STG", STG_CREATE Or STG_DIRECT Or
   STG_WRITE Or STG_SHARE_EXCLUSIVE)
```

Now we open a new stream within the storage by using the **CreateStream** function, one of the methods of the Storage Object.

```
Set Stream = RootStorage.CreateStream
   ("StreamName", STG_CREATE Or STG_DIRECT Or
   STG_WRITE Or STG_SHARE_EXCLUSIVE)
```

The Stream Object's **Put** method is one of the ways to actually place information into the file. However, even in Direct mode, data is still buffered. (See STG_DIRECT in the *Access Flags* section for a discussion of Direct and Transacted modes.) Therefore, after changing the stream, we have to clear the buffers and force the information to disk with the **Flush** and **Commit** methods.

```
Dim Info as Variant
Info = "This string is the information to be
   inserted into the file. "
Stream.Put 0, Info
Info = "This will be added after the first
   sentence."
Stream.Put STG_SEEK_DONTMOVE, Info
Stream.Flush STG_DEFAULT
RootStorage.Commit STG_DEFAULT
```

You have now successfully created a compound document with the file name "c:\test.stg". This compound document has a single stream with the name "StreamName" which contains two single strings.

Finally, always delete Visual Basic objects to free any memory that might be used.

```
Set dwStorage1 = Nothing
Set Stream = Nothing
Set RootStorage = Nothing
```

Visual Basic ActiveX Storage Control Example

The Storage Control takes advantage of the ability of Visual Basic to work with objects by creating two different types of objects which you can use to access any number of storages and streams. This simple example illustrates how to create these objects and use them. First, you must place the ActiveX control on a form that is loaded at the time the following code is run.

Create objects for each storage or stream you will need.

```
Dim RootStorage as Object
Dim Stream as Object
```

Next, create a file on disk. The example uses the Storage Control's method **CreateStorageFile**, which returns a storage object representing the root storage. The first parameter is the filename and the second parameter is the access mode.

```
Set RootStorage = DwStorage1.CreateStorageFile
   ("C:\TEST.STG", STG_CREATE Or STG_DIRECT Or
   STG_WRITE Or STG_SHARE_EXCLUSIVE)
```

Now we open a new stream within the storage by using the **CreateStream** function, one of the methods of the Storage Object.

```
Set Stream = RootStorage.CreateStream
   ("StreamName", STG_CREATE Or STG_DIRECT Or
   STG_WRITE Or STG_SHARE_EXCLUSIVE)
```

The Stream Object's **Put** method is one of the ways to actually place information into the file. However, even in Direct mode, data is still buffered (See STG_DIRECT in the *Access Flags* section for a discussion of Direct and Transacted modes.) Therefore, after changing the stream, we have to clear the buffers and force the information to disk with the **Flush** and **Commit** methods.

```
Dim Info as Variant
Info = "This string is the information to be
   inserted into the file. "
Stream.Put 0, Info
Info = "This will be added after the first
   sentence."
Stream.Put STG_SEEK_DONTMOVE, Info
Stream.Flush STG_DEFAULT
RootStorage.Commit STG_DEFAULT
```

You have now successfully created a compound document with the file name "c:\test.stg". This compound document has a single stream with the name "StreamName" which contains two single strings.

Finally, always delete Visual Basic Objects to free any memory that they might use.

```
Set Stream = Nothing
   Set RootStorage = Nothing
```

Storage Component Methods (dwStg.dll)

Note for .Net users:

All function descriptions below use the Visual Basic standard of "Long" for a 32 bit integer. In .NET languages, the correct type would be "Integer" in VB.NET or "int" in C#. They also use the Visual Basic data type "Variant". The .NET equivalent of that type is "Object".

Note that .NET does not correctly recognize the Boolean type of the return value in OLE components. You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function. Convert is an object in the "System" namespace – see the .NET documentation for more information.

*ComponentObject.**AllocateMemoryHandle () as Long***

This creates a global sharable block of memory suitable for Structured Storages. The block of memory uses 0 bytes initially, but expands to accommodate any data written to the Structured Storage. You may have more than one memory handle open at a time. In the .NET environment, you can use the Marshal.AllocHGlobal() function instead – pass a zero for the length of the buffer to allocate (memory will be allocated as needed by the system).

```
 ' VB
Dim hGlobal as LonghGlobal =
   DwStg.AllocateMemoryHandle()
```

*ComponentObject.**CompressStorageFile (Filename as String)***

After heavy usage, Compound Files become fragmented and can contain many gaps that take up disk space. This method unfragments and compresses the Compound File. This method is time intensive, and should not be used often (certainly *not* every time the file is modified - when used on significantly compressed files it might actually make the file larger).

```
DwStg.CompressStorageFile
        ("C:\MyDocuments\report.doc")
```

ComponentObject.***ConvertToLocalTime (MeanTimeDate as Date) as Date***

This converts a variable in Universal Coordinated Time (UCT time or Greenwich Mean Time) to the local time. This is needed because the storage creation, last modify and last addition dates are all stored in UCT time to prevent confusion when the file is sent to other time zones.

```
' VB
Dim CreateDate as Date
Dim UCTDate as Date

UCTDate = DwStg.GetCreationDate ()
CreateDate = DwStg.ConvertToLocalTime UCTDate
MsgBox "The storage " & DwStg.Name & " was created:
       " & CStr (CreateDate)

' VB.NET
Dim CreateDate As DateTime
Dim UCTDate As DateTime

UCTDate = DwStg.GetCreationDate ()
CreateDate = DwStg.ConvertToLocalTime (UCTDate)
MessageBox.Show ("The storage " & DwStg.Name & "
       was created: " & CreateDate.ToString())

// C#
private DateTime CreateDate;
private DateTime UCTDate;

UCTDate = DwStg.GetCreationDate ();
CreateDate = DwStg.ConvertToLocalTime (UCTDate);
MessageBox.Show ("The storage " + DwStg.Name + "
       was created: " + CreateDate.ToString());
```

ComponentObject.***CreateStorageFile (Filename as String, AccessMode as Long) as dStorage***

Creates a new blank Compound File on the disk. The **Filename** parameter should contain the name (with full path, as necessary) of the file to create. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **CreateStorageFile** is successful, a dStorage Object is returned that represents the root directory of the Structured Storage. If not, an error is triggered. (See the *Possible Errors* section for a description of errors).

This method can be used to open files that are not in the Structured Storage format. Include STG_CONVERT as one of the access mode flags. The file will appear to have a single stream named "Contents", which will contain the entire file contents. Simply reading from the file will cause no changes, but modifying anything will change the file into Structured Storage format permanently.

```
' VB
Dim dwStg as New dwStorage
Dim RootStorage as dStorage

' Create a file temp.stg if it does not exist
' or open it if it does.  Let other programs
' access the file as well.
Set RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", STG_CREATE Or STG_DIRECT Or
        STG_READWRITE Or STG_SHARE_DENYNONE)
' If the file temp.stg exists, overwrite it
' if it does not exist, create it
Set RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", STG_OVERWRITE Or STG_DIRECT
        Or STG_READWRITE Or STG_SHARE_EXCLUSIVE)

' VB.NET
Dim dwStg As dwStorage = New dwStorage()
Dim RootStorage As dStorage

' Create a file temp.stg if it does not exist
' or open it if it does.  Let other programs
' access the file as well.
```

```
RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", dwstg.STG_CREATE Or
        dwstg.STG_DIRECT Or dwstg.STG_READWRITE Or
        dwstg.STG_SHARE_DENYNONE)
' If the file temp.stg exists, overwrite it
' if it does not exist, create it
RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", dwstg.STG_OVERWRITE Or
        dwstg.STG_DIRECT Or dwstg.STG_READWRITE Or
        dwstg.STG_SHARE_EXCLUSIVE)
```

```
// C#
private dwStorage dwStg = new dwStorage();
private dStorage RootStorage;

// Create a file temp.stg if it does not exist
// or open it if it does.  Let other programs
// access the file as well.
RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", dwstg.STG_CREATE |
        dwstg.STG_DIRECT | dwstg.STG_READWRITE |
        dwstg.STG_SHARE_DENYNONE);
// If the file temp.stg exists, overwrite it
// if it does not exist, create it
RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", dwstg.STG_OVERWRITE |
        dwstg.STG_DIRECT | dwstg.STG_READWRITE |
        dwstg.STG_SHARE_EXCLUSIVE);
```

ComponentObject.***CreateStorageMemory***
(Name as String, AccessMode as Long, hGlobal as Long) as dStorage

Creates a new blank Structured Storage in memory at the location specified by the global memory handle **hGlobal**. (See **AllocateMemoryHandle** for information as to how to create one of these handles.) The storage is given the name found in the **Name** parameter, and is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. Memory storages are useful as a mode of organizing memory and allowing easy saving of data to disk. The Structured Storage initially utilizes no memory but expands to accommodate any data written to it.

If **CreateStorageMemory** is successful, a Storage Object is returned that represents the root directory of the Structured Storage. If not, an error is invoked. This storage can be copied to an existing storage on disk using the **CopyTo** or **MoveElementTo** methods.

```
' VB
Dim hGlobal as Long
Dim RootStgInMem as dStorage
Dim RootStgOnFile as dStorage

hGlobal = DwStg.AllocateMemoryHandle()
Set RootStgInMem = DwStg.CreateStorageMemory
        ("MemStg", STG_CREATE Or STG_READWRITE Or
        STG_DIRECT Or STG_SHARE_EXCLUSIVE, hGlobal)
' Now you can act on RootStgInMem just
' like any other storage.
' You can copy the entire contents to a file
' with a command similar to this:
RootStgInMem.CopyTo RootStgOnFile
' After you are done with the storage:
Set RootStgInMem = Nothing
DwStg.DeallocateMemoryHandle(hGlobal)


' VB.NET
Dim hGlobal As Long
Dim RootStgInMem As dStorage
Dim RootStgOnFile As dStorage

hGlobal = DwStg.AllocateMemoryHandle()
RootStgInMem = DwStg.CreateStorageMemory ("MemStg",
        dwstg.STG_CREATE Or dwstg.STG_READWRITE Or
        dwstg.STG_DIRECT Or
        dwstg.STG_SHARE_EXCLUSIVE, hGlobal)
' Now you can act on RootStgInMem just
' like any other storage.
' You can copy the entire contents to a file
' with a command similar to this:
RootStgInMem.CopyTo (RootStgOnFile)
' After you are done with the storage:
RootStgInMem.Finalize()
RootStgInMem = Nothing
DwStg.DeallocateMemoryHandle(hGlobal)


// C#
private int hGlobal;
```

```
private dStorage RootStgInMem;
private dStorage RootStgOnFile;

hGlobal = DwStg.AllocateMemoryHandle();
RootStgInMem = DwStg.CreateStorageMemory ("MemStg",
        dwstg.STG_CREATE Or dwstg.STG_READWRITE Or
        dwstg.STG_DIRECT Or
        dwstg.STG_SHARE_EXCLUSIVE, hGlobal);
// Now you can act on RootStgInMem just
// like any other storage.
// You can copy the entire contents to a file
// with a command similar to this:
RootStgInMem.CopyTo (RootStgOnFile);
// After you are done with the storage:
RootStgInMem.Finalize();
RootStgInMem = null;
DwStg.DeallocateMemoryHandle(hGlobal);
```

*ComponentObject.**DeallocateMemoryHandle (hGlobal as Long)***

This deallocates the block of memory allocated previously. Use this method after the Structured Storage is cleared (after the object is set to Nothing). In the .NET environment you can use the Marshal.FreeHGlobal() function instead.

```
' VB
DwStg.DeallocateMemoryHandle hGlobal
```

*ComponentObject.**EnableComponent (LicenseKey as String)***

This method allows you to use the Storage component in your own components that are used in Visual Basic or the .NET environment, such as ActiveX controls or ActiveX documents.

You first create a key by using the dwLicGen.exe program. This program will ask you to enter the compiled name of your component (not including a path, but including an extension). Each key is unique to a component name. If you change your component name, you'll have to create a new license key. After the program generates the key, you will have the option to copy it to the clipboard to post into your application.

You then call the **EnableComponent** method before you access any of the component's methods. It is best placed in your control's Initialize or Sub Main functions. If you ever unload or remove the last reference to the Storage component, you will have to go through this process again upon creating a new reference.

```
 ' VB
 Private Sub UserControl_Initialize()
' Your key will be different.
    DwStg.EnableComponent "2E3499248E322D3636853B53"
End Sub

 ' VB.NET // C#
 ' Your key will be different.
 DwStg.EnableComponent ("2E3499248E322D3636853B53")
```

*ComponentObject.**IsStorageFile (Filename as String) as Boolean***

Returns True if the file specified by the **Filename** parameter is a Compound File, False if it is not. If something goes wrong, an error is generated. Remember that non-Compound Files can still be opened as Compound Files. (This is done by using CreateStorageFile, specifying the file to be converted, and setting the STG_CONVERT flag. See the STG_CONVERT flag in the *Constants* section of this manual for more information.) ***Note that .NET does not correctly recognize the type of the return value***. You will need to convert the type back to a Boolean type – see the following code samples.

```
' VB
If (dwStg.IsStorageFile ("C:\report.doc")) Then
    MsgBox "Yes, report.doc is a storage file"
End If

' VB.NET
If (Convert.ToBoolean (dwStg.IsStorageFile
   ("C:\report.doc")) = True) Then
    MessageBox.Show ("Yes, report.doc is a storage
   file")
End If

// C#
if (Convert.ToBoolean (dwStg.IsStorageFile
   ("C:\report.doc")) == true)
{
```

```
    MessageBox.Show ("Yes, report.doc is a storage
   file");
}
```

*ComponentObject.**IsStorageMemory***
(hGlobal as Long) as Boolean

Returns True if the memory specified by the global memory handle **hGlobal** contains a Structured Storage, False if not. ***Note that .NET does not correctly recognize Boolean return types in OLE components.*** You will need to convert the type back to a Boolean type – see the following code samples.

```
If (dwStg.IsStorageMemory (hGlobal)) Then
    MsgBox "Yes, the memory referenced by  handle
   hGlobal does contain a storage"
EndIf

' VB.NET
If (Convert.ToBoolean (dwStg.IsStorageMemory
   (hGlobal)) = True) Then
    MessageBox.Show ("Yes, memory referenced by
      handle hGlobal does contain a storage ")
End If

// C#
if (Convert.ToBoolean (dwStg.IsStorageMemory
   (hGlobal)) == true)
{
    MessageBox.Show ("Yes, memory referenced by
      handle hGlobal does contain a storage ");
}
```

*ComponentObject.**IStreamToDStream***
(Long lpIStream)as dStream

This allows you to take a pointer to an OLE IStream interface and create a StorageTools dStream object from it. This is useful in cases where you receive a pointer to an IStream interface from a 3rd party component, but need StorageTools in order to access it.

```
' VB
Dim lpIStream as Long
Dim dStrm as dStream
Dim NumBytes as Long
```

```
' the pointer in lpIStream is provided from some
' 3rd party source
dStrm = IStreamToDStream (lpIStream)
' all dStream methods now work
NumBytes = dStrm.GetSize
```

ComponentObject.***OpenStorageFile (Filename as String, AccessMode as Long) as dStorage***

Opens a Compound File on disk. The **Filename** parameter contains the name (with full path, if needed) of the file to open. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **OpenStorageFile** is successful, a Storage Object is returned that represents the root directory of the Structured Storage. If not, an error is generated.

```
Dim dwStg as New dwStorage
Dim RootStorage as dStorage

' VB
' Open the file temp.stg for reading and
' writing and do not let anyone else access it.
Set RootStorage = dwStg.OpenStorageFile
        ("C:\temp.stg", STG_DIRECT or STG_READWRITE
        or STG_SHARE_EXCLUSIVE)
' Open the file temp.stg for reading only.
' Let anyone else read and write to the file.
Set RootStorage = dwStg.OpenStorageFile
        ("C:\temp.stg", STG_DIRECT or STG_READ or
        STG_SHARE_DENYNONE)

' VB.NET
Dim dwStg As dwStorage = New dwStorage()
Dim RootStorage As dStorage

' Open a file temp.stg for reading and
' writing and do not let anyone else access it.
RootStorage = dwStg.OpenStorageFile ("C:\temp.stg",
        dwstg.STG_DIRECT Or dwstg.STG_READWRITE Or
        dwstg.STG_SHARE_EXCLUSIVE)
' Open the file temp.stg for reading only.
' Let anyone else read and write to the file.
RootStorage = dwStg.OpenStorageFile ("C:\temp.stg",
        dwstg.STG_DIRECT Or dwstg.STG_READ Or
        dwstg.STG_SHARE_DENYNONE)
```

```
// C#
private dwStorage dwStg = new dwStorage();
private dStorage RootStorage;

// Open a file temp.stg for reading and
// writing and do not let anyone else access it.
RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", dwstg.STG_DIRECT |
        dwstg.STG_READWRITE |
        dwstg.STG_SHARE_EXCLUSIVE);
// Open the file temp.stg for reading only.
// Let anyone else read and write to the file
RootStorage = dwStg.CreateStorageFile
        ("C:\temp.stg", dwstg.STG_OVERWRITE |
        dwstg.STG_DIRECT | dwstg.STG_READWRITE |
        dwstg.STG_SHARE_DENYNONE);
```

ComponentObject.*OpenStorageMemory (Name as String, AccessMode as Long, hGlobal as Long) as dStorage*

Opens an already existing Structured Storage at the location in memory specified by the global memory handle **hGlobal**. It is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. The Structured Storage expands to accommodate any data written to it. If **OpenStorageMemory** is successful, a Storage Object is returned that represents the root directory of the Structured Storage. This storage can be copied to disk using the **CopyTo** or **MoveElementTo** methods.

```
' VB
Dim RootStgInMem as dStorage

' In this case, we probably get hGlobal from
' another task or module.
Set RootStgInMem = DwStg.OpenStorageMemory
        ("MemStg", STG_READWRITE Or STG_DIRECT Or
        STG_SHARE_EXCLUSIVE, hGlobal)
' Now you can act on RootStgInMem just
' like any other storage.
...
' After you are done with the storage:
Set RootStgInMem = Nothing
```

```
' VB.NET
Dim RootStgInMem As dStorage

' In this case, we probably get hGlobal from
' another task or module.
RootStgInMem = DwStg.CreateStorageMemory ("MemStg",
       dwstg.STG_READWRITE Or dwstg.STG_DIRECT Or
       dwstg.STG_SHARE_EXCLUSIVE, hGlobal)
' Now you can act on RootStgInMem just
' like any other storage.
...
' After you are done with the storage:
RootStgInMem.Finalize()
RootStgInMem = Nothing


// C#
private dStorage RootStgInMem;

RootStgInMem = DwStg.CreateStorageMemory ("MemStg",
       dwstg.STG_READWRITE Or dwstg.STG_DIRECT Or
       dwstg.STG_SHARE_EXCLUSIVE, hGlobal);
' Now you can act on RootStgInMem just
' like any other storage.
...
' After you are done with the storage:
RootStgInMem.Finalize();
RootStgInMem = null;
```

dStorage Object Methods (dwStg.dll)

*dStorage.**Commit***
(CommitFlags as Long)

If the storage is opened in **Transacted** mode (see Access Flags in the *Constants* section of this manual), this procedure makes permanent all changes made to the storage since it was opened or since the last **Commit**. It makes reverting any changes up to this point impossible. This call is also needed if the storage is opened in **Direct** mode to flush the buffer. **CommitFlags** describes how the storage should be committed (see the *Constants* section of this manual).

```
Dim RootStg as dStorage
' this performs a normal commit
RootStg.Commit 0

' VB.NET // C#
RootStg.Commit (0) ' "0" is shorthand for default
```

*dStorage.**CopyTo***
(DestinationStorage as dStorage)

Copies the contents of this storage to the already opened Storage Object passed to the **DestinationStorage** parameter.

```
Dim RootStg as dStorage
Dim backupStg as dStorage
' Copy the entire contents of one Storage file
' into the backupStg storage (which may be in
' another file or in memory).
' VB
RootStg.CopyTo backupStg

' VB.NET // C#
RootStg.CopyTo (backupStg)
```

dStorage.***CreateStorage (StorageName as String, AccessMode as Long) as dStorage***

Creates a sub-Storage in this Storage. The **Filename** parameter contains the name of the storage to be constructed. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **CreateStorage** is successful, a Storage Object is returned that represents the newly created storage. If not, an error is generated.

```
' VB
Dim RootStg as dStorage
Dim newStorage as dStorage
' Create a storage named DataBlocks in the
' RootStg storage if one does not exist, or
' open the storage named DataBlocks if it
' does.  This storage (like all non-root
' storages under OLE Structured Storage must
' be) cannot be accessed by other applications
' while the newStorage object exists.
Set newStorage = RootStg.CreateStorage
        ("DataBlocks", STG_CREATE or STG_READWRITE
        or STG_SHARE_EXCLUSIVE)
' When you are done with the storage, get rid
' of the object.
Set newStorage = Nothing
```

```
' VB.NET
Dim RootStg As dStorage
Dim newStorage As dStorage
newStorage = RootStg.CreateStorage ("DataBlocks",
        dwstg.STG_CREATE or dwstg.STG_READWRITE or
        dwstg.STG_SHARE_EXCLUSIVE)
newStorage.Finalize()
newStorage = Nothing
```

```
// C#
private dStorage RootStg;
private dStorage newStorage;
newStorage = RootStg.CreateStorage ("DataBlocks",
        dwstg.STG_CREATE | dwstg.STG_READWRITE |
        dwstg.STG_SHARE_EXCLUSIVE);
newStorage.Finalize();
newStorage = null;
```

dStorage.*CreateStream*
(StreamName as String, AccessMode as Long) as dStream

Creates a sub-stream in this storage. The **Filename** parameter contains the name of the stream to be constructed. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **CreateStream** is successful, a Stream Object is returned that represents the newly created stream. If not, an error is generated.

```
Dim RootStg as dStorage
Dim subStream as dStream
' Create a stream named Data1 in the RootStg
' storage if one does not exist, or open the
' stream named Data1 if it does.  This stream
' (like all streams under OLE Structured
' Storage must be) cannot be accessed by other
' applications while the newStream object
' exists.
Set subStream = RootStg.CreateStream ("Data1",
        STG_CREATE or STG_READWRITE or
        STG_SHARE_EXCLUSIVE)
' When you are done with the stream, get rid
' of the object.
Set newStream = Nothing

' VB.NET
Dim RootStg As dStorage
Dim subStream as dStream
newStorage = RootStg.CreateStream ("Data1",
        dwstg.STG_CREATE or dwstg.STG_READWRITE or
        dwstg.STG_SHARE_EXCLUSIVE)
newStream.Finalize()
newStream = Nothing

// C#
private dStorage RootStg;
private dStream subStream;
newStorage = RootStg.CreateStream ("Data1",
        dwstg.STG_CREATE | dwstg.STG_READWRITE |
        dwstg.STG_SHARE_EXCLUSIVE);
newStream.Finalize();
newStream = null;
```

dStorage.***DestroyElement (Name as String)***

Deletes the named storage or stream from this storage. If the element being deleted is a storage, then all the elements in that storage are also deleted. If there is any problem, an error is sent.

```
' Deletes the stream or storage named Data1
' in the RootStg storage.
' VB
RootStg.DestroyElement "Data1"

' VB.NET // C#
RootStg.DestroyElement ("Data1")
```

dStorage.***Directory (Index as Long, Type as Integer) as String***

This method can be used to obtain a catalogue of all storages and streams contained in this storage. Each call to **Directory** returns the name of the storage or stream specified by the **Index**. The **Type** integer is changed to reveal the type of the element: **STG_TYPE_STREAM** if the returned name represents a stream, **STG_TYPE_STORAGE** if the returned name represents a storage, and **STG_TYPE_NONE** if there are no more elements.

```
' VB
' enumerate through all the elements in the
' RootStg storage.  After the loop is done, the
' variable i will contain the count of elements
i = 0
Do
     ElementName = storage.Directory(i, FileType)
     If FileType = STG_TYPE_NONE Then Exit Do
     If FileType = STG_TYPE_STORAGE Then
         ' ElementName is a storage
     ElseIf FileType = STG_TYPE_STREAM Then
         ' ElementName is a stream
     End If
     i = i + 1
Loop

' VB.NET
i = 0
Do
     ElementName = storage.Directory(i, FileType)
     If FileType = dwstg.STG_TYPE_NONE Then Exit Do
     If FileType = dwstg.STG_TYPE_STORAGE Then
```

```
        ' ElementName is a storage
    ElseIf FileType = dwstg.STG_TYPE_STREAM Then
        ' ElementName is a stream
    End If
    i = i + 1
Loop

' C#
i = 0;
while (true)
{
    ElementName = storage.Directory(i, FileType)
    if (FileType == dwstg.STG_TYPE_NONE)
    {
        break;
    }
    if (FileType == dwstg.STG_TYPE_STORAGE)
    {
        ' ElementName is a storage
    }
    else if (FileType == dwstg.STG_TYPE_STREAM)
    {
        ' ElementName is a stream
    }
    i = i + 1;
}
```

dStorage.***EnumDirectory (RelativePos as Long, Type as Integer) as String***

This method can be used to obtain a catalogue of all storages and streams contained in this storage. It is faster than the normal **Directory** method, but it is less flexible. Each call to **EnumDirectory** returns the name of the first storage or stream if **RelativePos** is equal to 0, or the next storage or stream if **RelativePos** is not. The **Type** integer is changed to reveal the type of the element: **STG_TYPE_STREAM** if the returned name represents a stream, **STG_TYPE_STORAGE** if the returned name represents a storage, and **STG_TYPE_NONE** if there are no more elements.

```
' VB
' enumerate through all the elements in the
' RootStg storage.  After the loop is done, the
' variable i will contain the count of elements
i = 0
ElementName = storage.EnumDirectory(0, FileType)
Do
```

```
      If FileType = STG_TYPE_NONE Then Exit Do
      If FileType = STG_TYPE_STORAGE Then
          ' ElementName is a storage
      ElseIf FileType = STG_TYPE_STREAM Then
          ' ElementName is a stream
      End If
      ElementName = storage.EnumDirectory(1,
        FileType)
      i = i + 1
Loop
```

```
' VB.NET
i = 0
ElementName = storage.EnumDirectory(0, FileType)
Do
      If (FileType = dwstg.STG_TYPE_NONE) Then Exit
        Do
      If (FileType = dwstg.STG_TYPE_STORAGE) Then
          ' ElementName is a storage
      ElseIf (FileType = dwstg.STG_TYPE_STREAM) Then
          ' ElementName is a stream
      End If
      ElementName = storage.EnumDirectory(1,
        FileType)
      i = i + 1
Loop
```

```
// C#
i = 0;
ElementName = storage.EnumDirectory(0, FileType);
while (true)
{
      if (FileType == dwstg.STG_TYPE_NONE)
      {
          break;
      }
      if FileType == dwstg.STG_TYPE_STORAGE)
      {
          ' ElementName is a storage
      }
      else if (FileType == dwstg.STG_TYPE_STREAM)
      {
          ' ElementName is a stream
      }
      ElementName = storage.EnumDirectory(1,
        FileType);
      i = i + 1;
}
```

*dStorage.**Finalize ()***

This clears any internal buffers and references within the dStorage object. This method should be called before setting the dStorage object to "Nothing" in any .NET project. It is not neccessary to call this in Visual Basic 6 or any other programming environment that does not use garbage collection.

```
' VB.NET
Dim dStg as dStorage

' after you are done using dStg
dStg.Finalize()
dStg = Nothing

// C#
' after you are done using dStg
dStg.Finalize();
dStg = null;
```

*dStorage.**GetClass (CLSIDptr as Long) as Long***

Returns a pointer to a block of memory containing the CLSID associated with the storage.

```
Dim myCLSID As CLSID
Dim tempAddr As Long

tempAddr = dwGetAddressForRecord (myCLSID.part1)
Storage.GetClass tempAddr
Debug.Print "clsid 1 "; Hex$(myCLSID.part1)
```

*dStorage.**GetCreationDate () as Date***

Obtains the date this storage was created. Due to certain limitations in Windows, the date must be between the years 1980 and 2107.

```
' VB
Dim dt as Date
dt = Storage.GetCreationDate()

' VB.NET
Dim dt as DateTime
dt = Storage.GetCreationDate()

// C#
DateTime dt;
dt = Storage.GetCreationDate();
```

*dStorage.**GetIStorage () as Long***

GetIStorage returns a pointer to the IStorage object inside the dStorage. This can be useful with components that can directly access an IStorage interface, or when you are dealing with OLE interfaces that need an IStorage interface.

```
' Get a direct pointer to the IStorage interface
Dim ptrIStorage as Long
ptrIStorage = Storage.GetIStorage ()
```

*dStorage.**GetLastModifyDate () as Date***

Obtains the date this storage was last changed in any way. Due to certain limitations in Windows, the date must be between the years 1980 and 2107.

```
' VB
Dim dt as Date
dt = Storage.GetLastModifyDate()

' VB.NET
Dim dt as DateTime
dt = Storage.GetLastModifyDate()

// C#
DateTime dt;
dt = Storage.GetLastModifyDate();
```

*dStorage.**GetLastAccessDate () as Date***

Obtains the date this storage was last read. Not all file systems carry this information -- in which case **GetLastAccessDate** will return a zero as the last date of access. Due to certain limitations in Windows, the date must be between the years 1980 and 2107.

```
' VB
Dim dt as Date
dt = Storage.GetLastAccessDate()
If (dt = CDate(0)) Then
   MsgBox "Does not support Last Access time"
Else
   MsgBox "Last Access: " & CStr (dt)
End If

' VB.NET
Dim dt as DateTime
dt = Storage.GetLastAccessDate () If (dt =
  DateTime.FromOADate (0)) Then
```

```
    MessageBox.Show ("Does not support Last Access
  time")
Else
    MessageBox.Show ("Last Access:" &
  dt.ToString())
End If

// C#
private DateTime a dt;
dt = Storage.GetLastAccessDate ();
if (dt == DateTime.FromOADate (0))
{
   MessageBox.Show ("Does not support Last Access
  time");
}
else
{
   MessageBox.Show ("Last Access: " +
  dt.ToString());
}
```

*dStorage.**GetMIStorage () as Object***

GetMIStorage returns an IUnknown interface to the IStorage object inside the dStorage. This is more useful when using StorageTools in a language other than Visual Basic.

*dStorage.**GetMode () as Long***

GetMode returns a Long that contains all the flags that were used to open or create this storage.

```
' VB
' Did I open this storage with the STG_READWRITE
' flag?
Dim flags as Long
flags = Storage.GetMode ()
If (flags And STG_READWRITE) Then
    ' yes, I did open with the STG_READWRITE flag
Else
    ' no, I did not
End If

' VB.NET
Dim flags as Integer
flags = Storage.GetMode ()
If (flags And STG_READWRITE) Then
```

```
    ' yes, I did open with the STG_READWRITE flag
Else
    ' no, I did not
End If

// C#
Dim flags as Long
flags = Storage.GetMode ()
If (flags And STG_READWRITE) Then
    ' yes, I did open with the STG_READWRITE flag
Else
    ' no, I did not
End If
```

dStorage.*LoadObject (OLEobject as Object)*

LoadObject will ask the object contained in the OLEobject parameter to load itself from the data contained in this storage using its IPersistStorage interface. If the object does not support that interface, or if the storage does not conain valid persisted data, a Visual Basic error will result.

```
' See PutObject for a sample
```

dStorage.*MoveElementTo (ElementName as String, NewName as String, DestStorage as dStorage, MoveFlags as Long)*

Copies or moves an element from within this storage to the already open storage object passed to the **DestStorage** parameter. The element can be renamed by putting a different name in **NewName**. If **NewName** is **Null**, then the old name is kept. If **MoveFlags** is **STG_MOVEMOVE** then the original is erased (a move); if **MoveFlags** is **STG_MOVECOPY** then the original is not erased (a copy).

```
' VB
' Copy the stream Data1 in the RootStg storage
' into the storage named DataBackup.  Keep the
' same filename.
RootStg.MoveElementTo "Data1", vbNullString,
   DataBackup, STG_MOVECOPY
' Move the stream Data1 in the RootStg storage
' into the storage named DataBackup.  Change
' the filename to Yesterday1.
RootStg.MoveElementTo "Data1", "Yesterday1" ,
   DataBackup, STG_MOVEMOVE
```

```
' VB.NET
RootStg.MoveElementTo ("Data1", "Yesterday1" ,
   DataBackup, dwStg.STG_MOVEMOVE)
// C#
RootStg.MoveElementTo ("Data1", "Yesterday1" ,
   DataBackup, dwStg.STG_MOVEMOVE);
```

dStorage.***OpenStorage (StorageName as String, AccessMode as Long) as dStorage***

Opens a storage that is in the same file as this storage. The **Filename** parameter contains the name (and path, as necessary) of the storage to open. Like changing directories in DOS, the path can be relative to the root (with a leading backslash ("\")) or relative to this storage (without the leading backslash). The storage is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **OpenStorage** is successful, a Storage OLE Object is returned that represents the newly opened storage. If not, an error is generated.

```
Dim RootStg as dStorage
Dim subStorage as dStorage
' Open a storage named DataBlocks in the
' RootStg storage.  This storage (like all
' non-root storages under OLE Structured
' Storage must be) cannot be accessed by other
' applications while the subStorage object
' exists.
Set subStorage = RootStg.OpenStorage ("DataBlocks",
        STG_DIRECT or STG_READWRITE or
        STG_SHARE_EXCLUSIVE)
' When you are done with the storage, get rid
' of the object.
Set subStorage = Nothing

' VB.NET
Dim RootStg as dStorage
Dim subStorage as dStorage
subStorage = RootStg.OpenStorage ("DataBlocks",
        dwStg.STG_DIRECT Or dwStg.STG_READWRITE Or
        dwStg.STG_SHARE_EXCLUSIVE)
' When you are done with the storage, get rid
' of the object.
```

```
subStorage.Finalize()
subStorage = Nothing
// C#
private dStorage RootStg;
private dStorage subStorage;
subStorage = RootStg.OpenStorage ("DataBlocks",
        dwStg.STG_DIRECT | dwStg.STG_READWRITE |
        dwStg.STG_SHARE_EXCLUSIVE);
' When you are done with the storage, get rid
' of the object.
subStorage.Finalize();
subStorage = null;
```

dStorage.OpenStream (StreamName as String, AccessMode as Long) as dStream

Opens a sub-stream of this storage. The **Filename** parameter contains the name of the stream to open. The stream is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **OpenStream** is successful, a Stream OLE Object is returned that represents the newly opened stream. If not, an error is generated.

```
' VB
Dim RootStg as dStorage
Dim subStream as dStream
' Create a stream named Data1 in the RootStg
' storage if one does not exist, or open the
' stream named Data1 if it does.  This stream
' (like all streams under OLE Structured
' Storage must be) cannot be accessed by other
' applications while the newStream object
' exists.
Set subStream = RootStg.OpenStream ("Data1",
        STG_READWRITE or STG_SHARE_EXCLUSIVE)
' When you are done with the stream, get rid
' of the object.
Set subStream = Nothing

' VB.NET
Dim RootStg as dStorage
Dim subStream as dStream
subStream = RootStg.OpenStream ("Data1",
        dwStg.STG_READWRITE Or
        dwStg.STG_SHARE_EXCLUSIVE)
```

```
' When you are done with the stream, get rid
' of the object.
subStorage.Finalize()
subStream = Nothing

// C#
private dStorage RootStg;
private dStream subStream;
subStream = RootStg.OpenStream ("Data1",
        dwStg.STG_READWRITE |
        dwStg.STG_SHARE_EXCLUSIVE);
// When you are done with the stream, get rid
// of the object.
subStorage.Finalize();
subStream = null;
```

*dStorage.**PutObject (OLEobject as Object)***

PutObject will ask the object contained in the OLEobject parameter to save itself into this storage using its IPersistStorage interface. If the object does not support that interface a Visual Basic error will result.

```
' VB
' The ListView control supports the IPersistStorage
' interface.
stg.PutObject ListView1.object
stg.Commit 0
' Make a change to the ListView1 control to see
' the effect of loading it.
ListView1.View = lvwIcon
Debug.Print ListView1.View
' Load the original state of the object.
stg.LoadObject ListView1.object
' Now it will print the original state of the View
' property, not "lvwIcon".
Debug.Print ListView1.View
```

*dStorage.**RenameElement (OldName as String, NewName as String)***

Renames one of the storages or streams in this storage. Compound File names are limited to thirty-two characters in length.

```
' This renames the element named "Data" in
' storage RootStg to "ReserveData"
' VB
RootStg.RenameElement "Data", "ReserveData"
```

```
' VB.NET
RootStg.RenameElement ("Data", "ReserveData")
// C#
RootStg.RenameElement ("Data", "ReserveData");
```

*dStorage.**Revert ()***

When this method is called, if the storage is opened in **Transacted** mode, then all changes made to the storage since it was created or since the last **Commit** are discarded. **Revert** has no effect in **Direct** mode.

```
' Open an existing file in Transacted mode.
Set RootStg = dwStorage1.OpenStorageFile
   ("C:\file.stg", STG_TRANSACTED or STG_READWRITE
   or STG_SHARE_EXCLUSIVE)
' Create a new stream.
Set subStream = RootStg.CreateStream ("Data1",
   STG_CREATE or STG_DIRECT or STG_READWRITE or
   STG_SHARE_EXCLUSIVE)
' Now write the stream.  If the RootStg was
' Commited, it would contain a blank stream
' named "Data1".
subStream.Flush 0
Set subStream = Nothing
' This undoes all the changes I made to the
' RootStg storage - the stream subStream does
' not get written to the file.
RootStg.Revert

' VB.NET
' Open an existing file in Transacted mode.
RootStg = dwStorage1.OpenStorageFile ("C:\file.stg",
   dwStg.STG_TRANSACTED Or dwStg.STG_READWRITE Or
   dwStg.STG_SHARE_EXCLUSIVE)
' Create a new stream.
subStream = RootStg.CreateStream ("Data1",
   dwStg.STG_CREATE Or dwStg.STG_DIRECT Or
   dwStg.STG_READWRITE Or dwStg.STG_SHARE_EXCLUSIVE)
' Now write the stream.  If the RootStg was
' Commited, it would contain a blank stream
' named "Data1".
subStream.Flush (0)
subStream = Nothing
' This undoes all the changes I made to the
' RootStg storage - the stream subStream does
' not get written to the file.
```

```
RootStg.Revert ()

// C#
// Open an existing file in Transacted mode.
RootStg = dwStorage1.OpenStorageFile ("C:\file.stg",
   dwStg.STG_TRANSACTED | dwStg.STG_READWRITE |
   dwStg.STG_SHARE_EXCLUSIVE);
// Create a new stream.
subStream = RootStg.CreateStream ("Data1",
   dwStg.STG_CREATE | dwStg.STG_DIRECT |
   dwStg.STG_READWRITE | dwStg.STG_SHARE_EXCLUSIVE);
// Now write the stream.  If the RootStg was
// Commited, it would contain a blank stream
// named "Data1".
subStream.Flush (0);
subStream = null;
// This undoes all the changes I made to the
// RootStg storage - the stream subStream does
// not get written to the file.
RootStg.Revert ();
```

dStorage.***SetClass (CLSIDptr as Long)***

Sets the containing CLSID associated with the storage. **CLSIDptr** is a pointer to a block of memory containing the CLSID.

```
' there are other valid ways to define this as well
Type CLSID
    part1 As Long
    part2 As Integer
    part3 As Integer
    part4 As Byte
    part5 As Byte
    part6 As Byte
    part7 As Byte
    part8 As Byte
    part9 As Byte
    part10 As Byte
    part11 As Byte
End Type

Dim myCLSID As CLSID
Dim tempAddr as Long

myCLSID.part1 = &H11111111
myCLSID.part2 = &H2222
myCLSID.part3 = &H3333
```

```
myCLSID.part4 = &H44 ' and so on

tempAddr = dwGetAddressForRecord(myCLSID.part1)
Storage.SetClass tempAddr
```

dStorage.SetElementTimes (Element as String, CreationDate as Date, LastModifyDate as Date, LastAccessDate as Date)

Sets any or all of the dates for a sub-storage of the storage named in **Element** (streams do not carry any date information in the current implementation of OLE2). Use **Null** for any date you do not want changed. **LastAccessDate** is not saved on FAT or NTFS file systems, so changing it might not be useful. Due to certain limitations in Windows, the date must be between the years 1980 and 2107.

```
Dim Date1 as Date
Dim Date2 as Date

Date1 = Now
Date2 = Now
' Set the creation and last modify dates for
' the storage named "DataBlock1" in storage
' RootStg to the current date and time.  I only
' pass a zero for the last access time.
RootStg.SetElementTimes "DataBlock1", Date1, Date2,
   CDate(0)

' VB.NET
Dim Date1 as DateTime
Dim Date2 as DateTime
RootStg.SetElementTimes ("DataBlock1", Date1, Date2,
   DateTime.FromOADate (0))

// C#
private DateTime Date1;
private DateTime Date2;
RootStg.SetElementTimes ("DataBlock1", Date1, Date2,
   DateTime.FromOADate (0));
```

Summary Information Property Set Functions

These functions deal with the Summary Information stream in the relevant dStorage object.

Here is an example of how you would use these methods to edit a string in the Summary Information Property set:

```
' VB
' if the SummaryInformation stream exists, load
' its information
If (RootStorage.siOpenSummaryInfo = True) Then
    ' get the title
    titleString = RootStorage.siGetTitle()
    titleString = titleString & " version 2"
    ' set the new title
    RootStorage.siSetTitle titleString
    ' save the changes to the
    ' SummaryInformation stream
    RootStorage.siSaveSummaryInfo
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
     RootStorage.Commit STG_DEFAULT
End If

' VB.NET
' if the SummaryInformation stream exists, load
' its information
If (Convert.ToBoolean (RootStorage.siOpenSummaryInfo
   ()) = True) Then
    ' get the title
    titleString = RootStorage.siGetTitle()
    titleString = titleString & " version 2"
    ' set the new title
    RootStorage.siSetTitle (titleString)
    ' save the changes to the
    ' SummaryInformation stream
    RootStorage.siSaveSummaryInfo()
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
    RootStorage.Commit (dwStg.STG_DEFAULT)
End If

// C#
// if the SummaryInformation stream exists, load
// its information
if (Convert.ToBoolean (RootStorage.siOpenSummaryInfo
   ()) == true)
{
    // get the title
    titleString = RootStorage.siGetTitle();
    titleString = titleString + " version 2";
```

```
    // set the new title
    RootStorage.siSetTitle (titleString);
    // save the changes to the
    // SummaryInformation stream
    RootStorage.siSaveSummaryInfo();
    // just like any stream, you must commit the
    // root storage to actually write to disk
    RootStorage.Commit (dwStg.STG_DEFAULT);
}
```

dStorage.**siSetTitle (Title as String)**

Sets the title of the document.

dStorage.**siGetTitle () as String**

Returns the title of the document.

dStorage.**siSetSubject (Subject as String)**

Sets the subject of the document.

dStorage.**siGetSubject () as String**

Returns the subject of the document.

dStorage.**siSetAuthor (Author as String)**

Sets the author of the document.

dStorage.**siGetAuthor () as String**

Returns the author of the document.

dStorage.**siSetKeywords (Keyword as String)**

Sets key words related to the subject of the document. These keywords are typically used in search routines.

dStorage.**siGetKeywords () as String**

Returns keywords relating to the subject of the document.

dStorage.**siSetComments (Comment as String)**

Sets the comment field. This area might be used for making notes to oneself or others.

dStorage.**siGetComments () as String**

Returns the comment field.

dStorage.**siSetLastAuthor (LastAuthor as String)**

Sets the latest person to have modified the document.

dStorage.**siGetLastAuthor () as String**

Returns the last person who has modified the document.

dStorage.**siIncrementRevNum ()**

Increments the number of times a document has been revised.

dStorage.**siSetRevNum (TimesRevised as Long)**

Sets the number of times the document has been revised.

dStorage.**siGetRevNum () as Long**

Returns the number of times the document has been revised.

dStorage.**siStartEditTimer ()**

This starts an internal timer. It is useful for determining how long a document has been opened.

dStorage.**siAddEditTimerToTotal ()**

Adds the amount of time since siStartEditTimer was called to the field recording the total amount of time the document has been opened.

dStorage.**siGetTotalEditTime () as Long**

Returns the total amount of time a document has been open since it has been created, in minutes.

dStorage.**siSetTotalEditTime (Minutes as Long)**

Sets the total amount of time a document has been open, in minutes.

dStorage.**siRecordPrintDate ()**

Sets the field containing the last time this document was printed to the current date and time.

dStorage.**siGetLastPrintDate () as Date**

Returns the last time the document was printed.

dStorage.**siRecordCreateDate ()**

Sets the field containing the time the document was created to the current date and time.

dStorage.**siGetCreateDate () as Date**

Returns the time the document was created.

dStorage.**siRecordSaveDate ()**

Sets the field containing the last time this document was saved to the current date and time.

dStorage.**siGetLastSaveDate () as Date**

Returns the last time the document was saved.

dStorage.**siSetNumberOfPages (NumPages as Long)**

Sets the number of pages.

dStorage.**siGetNumberOfPages () as Long**

Returns the number of pages.

dStorage.**siSetNumberOfWords (NumWords as Long)**

Sets the number of words.

dStorage.siGetNumberOfWords () as Long

Returns the number of words.

dStorage.siSetNumberOfCharacters (NumChars as Long)

Sets the number of characters.

dStorage.siGetNumberOfCharacters () as Long

Returns the number of characters.

dStorage.siSetApplication (AppName as String)

Sets the name of the application that created the document.

dStorage.siGetApplication () as Long

Returns the title of the application that created the document.

dStorage.siSetTemplate (Template as String)

Sets the filename of the template used in the document.

dStorage.siGetTemplate () as String

Returns the filename of the template used in the document.

dStorage.siSetSecurity (SecurityLevel as Long)

Sets the recommended security level of the document. Note that this does not actually supply any security, but only serves as a reminder to the application that reads the file. The actions based on the values are as follows:

Value	Security Procedure
0	None.
1	Password protected. Do not allow viewing or editing of this document. Other programs cannot view or edit this document.
2	Read-Only recommend. The user will be warned if there is

	any attempt to edit the document.
3	Read-Only enforced. Does not allow any changes to the document.
4	Locked for Annotations. Does not allow any changes to the document.

dStorage.**siGetSecurity () as Long**

Returns the security level of the document. Your program should behave as follows:

Value	Security Procedure
0	None.
1	Password protected. Do not allow viewing or editing of this document unless you know what type of password protection is involved, and the password given is correct.
2	Read-Only recommend. Warn the user if there is any attempt to edit the document.
3	Read-Only enforced. Do not allow any changes to the document.
4	Locked for Annotations. Do not allow any changes to the document.

dStorage.**siOpenSummaryInfo () as Boolean**

Retrieves information from the SummaryInformation property set stream in the root storage of this Compound File. If there is no SummaryInformation property set, **siOpenSummaryInfo()** generates an error. Before the **siOpenSummaryInfo()** is called, using any function beginning with "siGet" will return the default value, which will be either zero or a blank string depending on the type. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

dStorage.**siSaveSummaryInfo ()**

Saves any changes to the SummaryInformation property set stream in the root storage. Just as with any stream, these changes are not actually written to disk until the root storage object's **Commit** method is called.

```
' VB
' Create a new storage and create a new SI
```

```
Dim RootStg as dStorage

Set RootStg = RootStorage.CreateStorage ("siTest",
  STG_CREATE or STG_READWRITE Or STG_DIRECT Or
  STG_SHARE_EXCLUSIVE)

RootStg.dsiSetNumWords 412
' You do not have to set all properties – when you
' save, all the rest of the properties will be
' saved with their default values.
RootStg.dsiSaveSummaryInfo
RootStg.Commit 0
Set RootStg = Nothing

' VB
Dim RootStg as dStorage

RootStg = RootStorage.CreateStorage ("siTest",
  dwStg.STG_CREATE Or dwStg.STG_READWRITE Or
  dwStg.STG_DIRECT Or dwStg.STG_SHARE_EXCLUSIVE)

RootStg.dsiSetNumWords (412)
RootStg.dsiSaveSummaryInfo()
RootStg.Commit (0)
RootStg.Finalize()
RootStg = Nothing

// C#
private dStorage RootStg;

dStg = RootStorage.CreateStorage ("siTest",
  dwStg.STG_CREATE | dwStg.STG_READWRITE |
  dwStg.STG_DIRECT | dwStg.STG_SHARE_EXCLUSIVE);

RootStg.dsiSetNumWords (412);
RootStg.dsiSaveSummaryInfo();
RootStg.Commit (0);
RootStg.Finalize();
RootStg = null;
```

Document Summary Information Property Set Functions

These functions deal with the Document Summary Information stream in the relevant dStorage object.

Here is an example of how you would use these methods to edit a string in the Document Summary Information Property set:

```
' VB
' if the DocumentSummaryInformation stream
' exists, load its information
If (RootStorage.dsiOpenSummaryInfo = True) Then
    ' get the name of the manager
    mngString = RootStorage.dsiGetManager()
    mngString = mngString & " and John Smith"
    ' set the new manager
    RootStorage.dsiSetManager mngString
    ' save the changes to the Document Summary
    ' Information stream
    RootStorage.dsiSaveSummaryInfo
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
    RootStorage.Commit STG_DEFAULT
End If

' VB.NET
If (Convert.ToBoolean
   (RootStorage.dsiOpenSummaryInfo ()) = True) Then
    ' get the name of the manager
    mngString = RootStorage.dsiGetManager()
    mngString = mngString & " and John Smith"
    ' set the new manager
    RootStorage.dsiSetManager (mngString)
    ' save the changes to the Document Summary
    ' Information stream
    RootStorage.dsiSaveSummaryInfo()
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
    RootStorage.Commit (STG_DEFAULT)
End If

// C#
if (Convert.ToBoolean
   (RootStorage.dsiOpenSummaryInfo()) == true)
{
    // get the name of the manager
    mngString = RootStorage.dsiGetManager();
    mngString = mngString + " and John Smith"
    // set the new manager
    RootStorage.dsiSetManager (mngString);
    // save the changes to the Document Summary
    // Information stream
```

```
    RootStorage.dsiSaveSummaryInfo();
    // just like any stream, you must commit the
    // root storage to actually write to disk
    RootStorage.Commit (STG_DEFAULT);
}
```

dStorage.**dsiSetScaleCrop (Scale as Boolean)**

Set this to True if this document is supposed to be scaled to the dimensions of the window, False if it is to be cropped to the dimensions of the window.

dStorage.**dsiGetScaleCrop () as Boolean**

Returns True if this document is supposed to be scaled to the current dimensions of the window, False if it is to be cropped to the dimensions of the window. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

dStorage.**dsiSetLinksUpToDate (Links as Boolean)**

Set this to True if all the links are up to date. Links may refer to internal links in the document or links to web sites depending upon the application.

dStorage.**dsiGetLinksUpToDate () as Boolean**

Returns True if all the links are up to date. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

dStorage.**dsiSetCategory (Category as String)**

Sets the category of the document.

dStorage.**dsiGetCategory () as String**

Returns the category of the document.

dStorage.**dsiSetPresentationTarget (Target as String)**

Sets the presentation target of the document. Used in Microsoft PowerPoint.

dStorage.**dsiGetPresentationTarget () as String**

Returns the presentation target of the document.

dStorage.**dsiSetManager (Manager as String)**

Sets the manager of the document writer.

dStorage.**dsiGetManager () as String**

Returns the manager of the document writer.

dStorage.**dsiSetCompany (Company as String)**

Sets the company with which the document is associated.

dStorage.**dsiGetCompany () as String**

Returns the company with which the document is associated.

dStorage.**dsiSetNumBytes (NumBytes as Long)**

Sets the size (in number of bytes) of the document. Usually refers to the section of the document edited by the user, not the entire size of the file.

dStorage.**dsiGetNumBytes () as Long**

Returns the size of the document in number of bytes.

dStorage.**dsiSetNumLines (NumLines as Long)**

Sets the number of lines in the document.

dStorage.**dsiGetNumLines () as Long**

Returns the number of lines in the document.

dStorage.**dsiSetNumParagraphs (NumParas as Long)**

Sets the number of paragraphs in the document.

dStorage.**dsiGetNumParagraphs () as Long**

Returns the number of paragraphs in the document.

dStorage.**dsiSetNumSlides (NumSlides as Long)**

Sets the number of slides in the document.

dStorage.**dsiGetNumSlides () as Long**

Returns the number of slides in the document.

dStorage.**dsiSetNumNotes (NumNotes as Long)**

Sets the number of notes in the document.

dStorage.**dsiGetNumNotes () as Long**

Returns the number of notes in the document.

dStorage.**dsiSetNumHiddenSlides (NumHiddenSlides as Long)**

Sets the number of hidden slides in the document.

dStorage.**dsiGetNumHiddenSlides () as Long**

Returns the number of hidden slides in the document.

dStorage.**dsiSetNumMMClips (NumClips as Long)**

Sets the number of multimedia clips in the document.

dStorage.**dsiGetNumMMClips () as Long**

Returns the number of multimedia clips in the document.

dStorage.**dsiOpenDocSummaryInfo () as Boolean**

Retrieves information from the Document Summary Information Property Set stream. If there is no Document Summary Information Property Set, **dsiOpenDocSummaryInfo** returns False. Before the **dsiOpenDocSummaryInfo** method is called, or if there is no Document Summary Information stream, using any "dsi" method beginning with "dsiGet " will return the default value, which will be either zero or a blank string depending upon the type. To make a new Document Summary Information stream where one does not yet exist, set the above properties to the values you want and then call the **dsiSaveDocSummaryInfo** method. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

dStorage.**dsiSaveDocSummaryInfo ()**

Saves any changes to the Document Summary Information Property Set stream. Just as with any stream, these changes are not actually written to disk until the root storage object's **Commit** method is called.

```
' Create a new storage and create a new Document SI
Dim RootStg as dStorage

Set RootStg = RootStorage.CreateStorage ("dsiTest",
  STG_CREATE or STG_READWRITE Or STG_DIRECT Or
  STG_SHARE_EXCLUSIVE)

RootStg.dsiSetNumBytes 1024
' You do not have to set all properties – when you
' save, all the rest of the properties will be
' saved with their default values.
RootStg.dsiSaveDocSummaryInfo


' VB.NET
Dim RootStg As dStorage

RootStg = RootStorage.CreateStorage ("dsiTest",
  dwStg.STG_CREATE Or dwStg.STG_READWRITE Or
  dwStg.STG_DIRECT Or dwStg.STG_SHARE_EXCLUSIVE)
```

```
RootStg.dsiSetNumBytes (1024)
RootStg.dsiSaveDocSummaryInfo()
RootStg.Commit (0)

// C#
private dStorage RootStg;

RootStg = RootStorage.CreateStorage ("dsiTest",
  dwStg.STG_CREATE | dwStg.STG_READWRITE |
  dwStg.STG_DIRECT | dwStg.STG_SHARE_EXCLUSIVE);

RootStg.dsiSetNumBytes (1024);
RootStg.dsiSaveDocSummaryInfo();
RootStg.Commit (0);
```

User Document Summary Information Property Set Functions

These functions deal with the user-defined portion of the Document Summary Information stream in the relevant dStorage object.

Here is an example of how you would use these methods to read and print a list of existing properties, and adding a number and a string into the Document Summary Information Property set:

```
' VB
Dim count as Long
Dim v as Variant
Dim i as Long

' if the DocumentSummaryInformation stream
' exists, load its information
count = 0
If (Stg.dsiOpenUserSummaryInfo = True) Then
    ' get the number of properties
    count = Stg.dsiUserCount
    ' print all the property names and data
    For i = 0 To count - 1
        Debug.Print "name:", Stg.dsiUserDirectory(i)
        Debug.Print "data:", Stg.dsiUserGet(i)
    Next i

    ' Add a property that contains a number
    v = 1234
    count = Stg.dsiUserAdd("NewEntry1", v)
```

```
    ' Add a property that contains a string
    v = "String Value"
    count = Stg.dsiUserAdd("NewEntry2", v)

    Stg.dsiSaveUserSummaryInfo
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
    Stg.Commit STG_DEFAULT
End If

' VB.NET
Dim count as Integer
Dim v as Object
Dim i as Integer

' if the DocumentSummaryInformation stream
' exists, load its information
count = 0
If (Convert.ToBoolean (Stg.dsiOpenUserSummaryInfo())
  = True) Then
    ' get the number of properties
    count = Stg.dsiUserCount()
    ' print all the property names and data
    For i = 0 To count - 1
        MessageBox.Show ("name:" &
  Stg.dsiUserDirectory(i) & " data:" &
  Stg.dsiUserGet(i))
    Next i

    ' Add a property that contains a number
    v = 1234
    count = Stg.dsiUserAdd("NewEntry1", v)
    ' Add a property that contains a string
    v = "String Value"
    count = Stg.dsiUserAdd("NewEntry2", v)

    Stg.dsiSaveUserSummaryInfo()
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
    Stg.Commit (STG_DEFAULT)
End If

// C#
int countv;
object v;
int i;
```

```
// if the DocumentSummaryInformation stream
// exists, load its information
count = 0;
if (Convert.ToBoolean (Stg.dsiOpenUserSummaryInfo())
   == true)
{
    // get the number of properties
    count = Stg.dsiUserCount();
    // print all the property names and data
    for (i = 0; i > count; i++)
    {
        MessageBox.Show ("name:" &
  Stg.dsiUserDirectory(i) & " data:" &
  Stg.dsiUserGet(i));
    }

    // Add a property that contains a number
    v = 1234;
    count = Stg.dsiUserAdd("NewEntry1", v);
    // Add a property that contains a string
    v = "String Value";
    count = Stg.dsiUserAdd("NewEntry2", v);

    Stg.dsiSaveUserSummaryInfo();
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
    Stg.Commit (STG_DEFAULT);
}
```

dStorage**.dsiUserSet (PropertyID as Long, Name as String, Data as Variant)**

Modifies the existing property specified by **PropertyID**. **Data** can be either an integer, a date, a string, or a boolean value. Regular integers are converted to long integers to match the existing standard. If **PropertyID** does not correspond to an existing property, an error is generated. Due to certain limitations in Windows, dates must be between the years 1980 and 2107. If you need to store a date outside this range, you can use a string.

dStorage.**dsiUserGet**
(PropertyID as Long) as Variant

Returns the data associated with the property specified by **PropertyID**.

dStorage.**dsiUserAdd**
(Name as String, Data as Variant) as Long

Add a new property to the end of the list of user-defined properties in this Document Summary Information stream. The new count of user-defined properties is returned. Due to certain limitations in Windows, dates must be between the years 1980 and 2107 – if you have to store a date that is outside this range, you can use a string.

```
' VB
' Add a string
Dim v as Variant
v = "New String Data"
count = RootStg.dsiUserAdd ("value name", v)

' VB.NET
Dim v as Object
v = "New String Data"
count = RootStg.dsiUserAdd ("value name", v)

// C#
object v;
v = "New String Data";
count = RootStg.dsiUserAdd ("value name", v);
```

dStorage.**dsiUserCount () as Long**

Returns the number of user-defined properties in this Document Summary Information stream.

dStorage.**dsiUserDelete**
(PropertyID as Long) as Long

This deletes the specified property. After the deletion, all properties are renumbered to remove the empty space. The new count of user-defined properties is returned.

```
' Delete the last property
count = dwStg.dsiUserCount
count = dwStg.dsiUserDelete(count - 1)
```

dStorage.**dsiUserDirectory (PropertyID as Long) as String**

Returns the name of the specified property.

dStorage.**dsiOpenUserSummaryInfo () as Boolean**

Retrieves information from the user-defined section of the Document Summary Information Property Set stream. If there is no Document Summary Information Property Set, **dsiOpenUserSummaryInfo** returns False. To make a new Document Summary Information stream where one does not yet exist, use **dsiUserAdd** to create the properties you want and then call the **dsiSaveUserSummaryInfo** method. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

dStorage.**dsiSaveUserSummaryInfo ()**

Saves any changes to the user-defined section of the Document Summary Information Property Set stream. Just as with any stream, these changes are not actually written to disk until the root storage object's **Commit** method is called.

```
' VB
' Create a new Document SI
 Dim count As Long
 Dim v As Variant
 Dim RootStg as dStorage

 v = True ' Add a boolean property
 count = dwStg.dsiUserAdd("FirstElement", v)
 RootStg.dsiSaveUserSummaryInfo
 RootStg.Commit 0

' VB.NET
' Create a new Document SI
 Dim count As Integer
 Dim v As Object
 Dim RootStg as dStorage

 v = True ' Add a boolean property
 count = RootStg.dsiUserAdd("FirstElement", v)
 RootStg.dsiSaveUserSummaryInfo
 RootStg.Commit (0)
```

```
// C#
// Create a new Document SI
 int count;
 object v;
 dStorage RootStg;

 v = true; // Add a boolean property
 count = RootStg.dsiUserAdd("FirstElement", v);
 RootStg.dsiSaveUserSummaryInfo();
 RootStg.Commit (0);
```

dStorage Object Properties (dwStg.dll)

*dStorage**.IsValid***

A Boolean value, True if the dStorage represents a storage and False if not. A dStorage variable is not valid until it has been set using a successful call to the CreateStorageFile, OpenStorageFile, CreateStorageMemory, OpenStorageMemory, CreateStorage or OpenStorage methods. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert this property back to a Boolean type using the Convert.ToBoolean () function.

*dStorage**.Name***

The name of the dStorage. Begins as the name of the storage itself. For root storages, the storage name is the same as the filename. Use the **RenameElement** method to rename the actual streams and storages, or Windows file system methods to rename the file.

```
' If I have the object, I might still need the
' name for some function calls.
RootStg.RenameElement RootStg.Name, "NewName"
```

dStream Object Methods (dwStg.dll)

*dStream.**Finalize ()***

This clears any internal buffers and references within the dStream object. This method should be called before setting the dStream object to "Nothing" in any .NET project. It is not neccessary to call this in Visual Basic 6 or any other programming environment that does not use garbage collection.

```
' VB.NET
Dim dStrm as dStream
' after you are done using dStrm
dStrm.Finalize()
dStrm = Nothing

// C#
' after you are done using dStrm
dStrm.Finalize();
dStrm = null;
```

*dStream.**Flush***
(CommitFlags as Long)

In the current implementation of OLE2, streams can only be opened in Direct mode. **Flush** flushes any internal buffers (although this only speeds up what would have been done by a few seconds in any case). **CommitFlags** describes how the stream should be committed (see the *Constants* section of this manual).

```
' Perform a normal flush of buffersStream.Flush 0 '
  VB
Stream.Flush (0) ' VB.NET // C#
```

*dStream.**Get***
(SeekPosition as Long, Buffer as Variant)

Reads information in Visual Basic Binary form from this stream. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to read at the location specified by the last **Seek** or at where the last **Get** or **Put** operation concluded. The amount of data read depends upon the size and type of the variant **Buffer.**

```
' VB
' Read in two 10-character strings from the start
```

```
' of the stream (the first 20 bytes). No matter what
' the data originally meant, Get will interpret it
' as the type of the Variant passed to it.
Dim Buffer as Variant
' Pre-set the variant to 10 characters
Buffer = String (10, 0)
Stream.Get 0, Buffer
Debug.Print Buffer
' Now get the second 10 character string
Stream.Get SEEK_DONTMOVE, Buffer
Debug.Print Buffer

' VB.NET
Dim Buffer as Object
' Pre-set the variant to 10 characters
Buffer = String (10, 0)
Stream.Get 0, Buffer
MessageBox.Show (Buffer.ToString())
' Now get the second 10 character string
Stream.Get (dwStg.SEEK_DONTMOVE, Buffer)
MessageBox.Show (Buffer.ToString())

// C#
object Buffer;
// Pre-set the variant to 10 characters
Buffer = "          "; // 10 spaces
Stream.Get (0, Buffer);
MessageBox.Show (Buffer.ToString());
// Now get the second 10 character string
Stream.Get (dwStg.SEEK_DONTMOVE, Buffer);
MessageBox.Show (Buffer.ToString());
```

*dStream.**GetBlock***
(SeekPosition as Long, NumOfBytes as Long) as Long

GetBlock can be used to get a large block of data from a stream. This may be useful if you are trying to manipulate data in a stream larger than the limits of what a string or array can hold. It returns a pointer to an internally allocated block holding the data from the stream starting at **SeekPosition**. If there is any problem in allocating the block, the return value is 0. If **NumOfBytes** is larger than the number of bytes in the stream, the buffer from the end of the data on is filled with zeroes. The internal buffer is automatically deleted when the stream object is destroyed (for example, when it is set to Nothing or set to another stream), so be sure to keep the stream object as long as you are accessing the memory block. Using this method will release any previously allocated buffer.

```
' Points to the block of memory where I load
' the data from the stream.
Dim BlockPtr As Long
' Byte array where I will copy the data.  This
' will make it easier to manage in Visual Basic
Dim datablock() As Byte

' Get all the data in the stream.
BlockPtr = stream.GetBlock(0, stream.GetSize())
' Adjust the size of the byte array to that of
' the block I just read.
ReDim datablock(stream.GetSize())

' Copy the data from the pointer to the byte
' array.  The "stg*" functions are part of the
' dwAddr32.dll support library.
stgCopyDataBynum BlockPtr,
   stgGetAddressForObject(datablock(0)),
   stream.GetSize()

' Now all the data in the stream is in the
' datablock byte array.  After you are done
' with the data, write it back:
 stream.PutBlock 0, stream.GetSize(),
         stgGetAddressForObject(datablock(0))
```

*dStream.**GetBlockCopy (Long SeekPosition, Long NumBytes, Long BlockPtr)***

GetBlockCopy can be used to get a large block of data from a stream. This is different than **GetBlock** in that it can copy the data into a block of memory that already exists. The pointer to this block of memory is passed in the **BlockPtr** parameter. Be sure that there is sufficient space in memory for the number of bytes you are reading from the stream.

*dStream.**GetIStream () as Long***

GetIStream returns a pointer to the IStream object inside the dStream. This can be useful with components that can directly access an IStream interface, or when you are dealing with OLE interfaces that need an IStream interface.

```
' Get a direct pointer to the IStream interface
Dim ptrIStream as Long
ptrIStream = Stream.GetIStream ()
```

*dStream.**GetMIStream () as Object***

GetMIStream returns an IUnknown interface to the IStream object inside the dStream. This is more useful when using StorageTools in a language other than Visual Basic.

*dStream.**GetMode () as Long***

GetMode returns a long that contains all the flags that were used to open or create this stream.

```
' Did I open this stream with the STG_READWRITE
' flag?
Dim flags as Long
flags = Stream.GetMode ()
If (flags And STG_READWRITE) Then
    ' yes, I did open with the STG_READWRITE flag
Else
    ' no, I did not
End If
```

*dStream.**GetObject () as Object***

GetObject returns an object that was previously saved to this particular stream with **PutObject** or with the object's own persistence mechanism. This supports objects with an IPersistStream or IPersistStreamInit interface.

```
' Get the object in stream ObjStream
Dim myObject as Object
Set myObject = ObjStream.GetObject ()
```

*dStream.**GetPicture () as Picture***

GetPicture is used to read a picture from the stream. See **PutPicture** for information on how the picture is stored into a stream. **GetPicture** is compatible with the Picture property of forms, picture boxes, image boxes and other controls.

```
' Get the picture in stream PicStream
' Picture1 is a picture box control
Picture1.Picture = PicStream.GetPicture ()
```

*dStream.**GetRecord (RecordNumber as Long, RecordSize as Long, RecordAddress as Long)***

Used for saving User Defined Types (UDT). Reads a block of information from the record **RecordNumber** of size **RecordSize**. The **RecordNumber** parameter specifies the number of records from the start of the file, beginning with zero. All UDT's in the stream must be of the same size in order to prevent corruption of data. Records containing variable length strings will not work. Use the constant **SEEK_DONTMOVE** in the **RecordNumber** parameter to read at the location specified by the last **Seek** or where the last **GetRecord** or **PutRecord** terminated. If you do use **Seek**, be sure to seek to a whole multiple of the size of the record. You can get the address of a record by using the **stgGetAddressForRecord** function from the DWADDR32.DLL. In .NET

```
' VB
' Declare my type
 Private Type TypeX
    a As Integer
    b As String * 10
    c As Date
 End Type

 Dim Stream as dStream
 Dim RecordA As TypeX
 Dim RecordB As TypeX
 Dim adr As Long
 Dim tst As Integer
```

```
Set Stream = RootStg.CreateStream("UDT", STG_CREATE
  Or STG_DIRECT Or STG_SHARE_EXCLUSIVE Or
  STG_READWRITE)
RecordA.a = 500
RecordA.b = "qwertyuiop"
RecordA.c = Now – 1
' To get the correct address for a UDT in 32-bit
' Visual Basic, you must get the address for the
' first element.  As it is difficult to get the
' address for a string in 32-bit Visual Basic, you
' cannot make a string the first element in your
' UDT.  See chapter 15 of the Visual Basic
' Programmers Guide to the Win32 API by Daniel
' Appleman for more information.
adr = stgGetAddressForRecord(RecordA.a)
' Put the first record.  Note that I use the LenB
' command to get the correct length
Stream.PutRecord 0, LenB(RecordA), adr
RecordA.a = 600
RecordA.b = "abcdefghij"
RecordA.c = Now
' Put the second record.
adr = stgGetAddressForRecord(RecordA.a)
Stream.PutRecord 1, LenB(RecordA), adr
RecordA.a = 700
RecordA.b = "lmnopqrstu"
RecordA.c = Now + 1
adr = stgGetAddressForRecord(RecordA.a)
' Put the third record
Stream.PutRecord 2, LenB(RecordA), adr
Stream.Flush 0
RootStg.Commit 0

' Put some dummy values so we can see the changes
RecordB.a = 1
RecordB.b = ".........."
RecordB.c = 0
adr = stgGetAddressForRecord(RecordB.a)
' Get the first record
Stream.GetRecord 0, LenB(RecordB), adr

' This prints "500","qwertyuiop" and the date
Debug.Print xy.a, xy.b, xy.c

RecordB.a = 1
RecordB.b = ".........."
RecordB.c = 0
```

```
' Get the third record.  This shows the random-
' access ability of the method.
Stream.GetRecord 2, LenB(RecordB),
  stgGetAddressForRecord(RecordB.a)

' This prints "700","lmnopqrstu"and tomorrow's
' date
Debug.Print RecordB.a, RecordB.b, RecordB.c
```

*dStream.**GetSize () as Long***

Returns the size, in bytes, of this stream.

```
' Get the size in number of bytes of a stream
 NumBytes = PicStream.GetSize ()
```

*dStream.**GetString (StringData as String)***

Obtains a string from the Visual Basic Sequential formatted stream. The string returned is changed to the correct size. You will need to convert the string to the correct type using Visual Basic conversion functions. The seek pointer should be pointing to the start of the item to retrieve. After **GetString** is done the seek pointer will point to the next item.

```
' VB
' If the next data item in the stream is the
' integer "123", the statement
Stream.GetString InputString
' will return the string “123”, which you can
' convert to an integer with:
InputInteger = CInt(InputString)

' VB.NET
Stream.GetString (InputString)
InputInteger = Convert.ToInt32 (InputString)

// C#
Stream.GetString (InputString);
InputInteger = Convert.ToInt32 (InputString);
```

*dStream.**LoadObject***
(PersistedObject as Object)

LoadObject tells the **PersistedObject** object to load itself with the data that was previously saved in this particular stream with **PutObject** or with the object's own persistence mechanism. This is different than **GetObject** in that **GetObject** creates a new object and returns it. You would use this when you deal with the "Object" property of an object, like an ActiveX control. This supports objects with an IPersistStream or IPersistStreamInit interface.

```
' VB
' There is a control called "NumberControl" already
' on the form.
NumberControl.SetNumber = 123
Stream.LoadObject NumberControl.Object
' This will not print "123", but whatever state was
' previously saved in this stream.
Debug.Print NumberControl.GetNumber
```

*dStream.**Put***
(SeekPosition as Long, Buffer as Variant)

Writes information to this stream in the same way Visual Basic writes binary data. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to write at the location specified by the last **Seek** or at where the last **Get** or **Put** concluded. The amount of information written depends on the length of the data in the variant **Buffer**. If the data is written past the end of the stream, the stream automatically expands.

```
' VB
' Save a 10-character string at the start of
' the stream.
Dim Buffer as Variant
Buffer = "abcdefghij"
Stream.Put 0, Buffer
' Save a long integer to the place in the
' stream where the seek pointer currently is.
Buffer = 123456
Stream.Put SEEK_DONTMOVE, Buffer

' VB.NET
Dim Buffer as Object
Buffer = "abcdefghij"
```

```
Stream.Put (0, Buffer)
Buffer = 123456
Stream.Put (SEEK_DONTMOVE, Buffer)

// C#
object Buffer;
Buffer = "abcdefghij";
Stream.Put (0, Buffer);
Buffer = 123456;
Stream.Put (SEEK_DONTMOVE, Buffer);
```

*dStream.**PutBlock (SeekPosition as Long, NumOfBytes as Long, Pointer as Long)***

PutBlock is used to write a large block of data to a stream. This may be useful if you are trying to manipulate data larger than the limits of what a string or array can hold. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to write to the location specified by the last **Seek** or at where the last stream method concluded. **NumOfBytes** specifies the number of bytes, starting at memory location **Pointer**, that are written to the stream. The dwAddress .DLL's come with functions for obtaining pointers and manipulating data within the block.

See **GetBlock** for a sample.

*dStream.**PutObject (PersistedObject as Object)***

PutObject will persist (save) an object to a particular stream. You can only save one item per stream, and you should not write any other information to that stream. This function supports objects with an IPersistStream or IPersistStreamInit interface.

```
' VB
' Save a PersistTest object in stream ObjStream
Dim pn As New PersistTest
Dim dwStg1 As New dwStorage
Dim root As dStorage
Dim strm As dStream

' Set the state of the object.
pn.SetNumber 555
```

```
Set root =
   dwStg1.CreateStorageFile("C:\persist.stg", flags)
Set strm = root.CreateStream("number", flags)
strm.PutObject pn
' Make sure the information is flushed sometime
' after writing.
strm.Flush 0
root.Commit 0

Call strm.Seek(0, STG_STREAM_SEEK_SET)
' Change the state of the object.  If we are not
' loading correctly, 333 will show instead of 555.
pn.SetNumber 333
' Load the data back into the same object.
Set pn = strm.GetObject
Debug.Print "State of object pn:"; pn.GetNumber

Set strm = Nothing
Set root = Nothing
```

*dStream.**PutPicture (Pic as Picture)***

PutPicture is used to write a picture to the stream. The picture will take up the entire stream, and it will not be possible to write other information to the stream without corrupting the picture. This is compatible with the Picture property of forms, picture boxes, image boxes and other controls.

```
' Picture1 is a picture box control
PicStream.PutPicture Picture1.Picture
```

*dStream.**PutRecord***
(RecordNumber as Long, RecordSize as Long, RecordAddress as Long)

Writes a block of information to the record **RecordNumber** of size **RecordSize**. The **RecordNumber** parameter specifies the number of records from the start of the file, beginning with zero. All records in the stream must be the same size and must match the size of the record exactly in order to prevent the corruption of data. Use the constant **SEEK_DONTMOVE** to write at the location specified by the last **Seek** or at where the last **GetRecord** or **PutRecord** concluded. If you do use **Seek**, be sure to seek to a whole multiple of the size of the record. You can obtain the address of a record by using the **GetAddressForRecord** function from the DWADDR32.DLL. If the user-defined-type you are using contains any strings, be sure to use the "LenB" Visual Basic command to get the right size of the record.

See GetRecord for an example.

*dStream.**PutString***
(StringData as String, VarType as Integer, Comma as Integer)

This writes a string to the stream in Visual Basic Sequential format. **VarType** should contain the type of the variable (obtained from Visual Basic with the VarType() function). If **Comma** is True, then commas are used to separate items; if not, then a Line Feed/Carriage Return is used. The stream automatically expands if the data is written past the end of the file.

```
' VB
' Write a string to the stream, with a comma
' afterwards
Dim StrData as String
StrData = "first string"
Stream.PutString StrData, VarType(StrData), -1
' Write a number without a comma
StrData = CStr(123)
Stream.PutString StrData, VarType(123), 0

' VB.NET
Dim StrData as String
StrData = "first string"
Stream.PutString (StrData, VarType(StrData), -1)
```

```
StrData = 123.ToString();
Stream.PutString (StrData, VarType(123), 0)
// C#
string StrData;
StrData = "first string";
Stream.PutString (StrData, VarType(StrData), -1);
StrData = 123.ToString();
Stream.PutString StrData, VarType(123), 0);
```

dStream.Seek (Position as Long, SeekFlag as Long) as Long

Sets the position at which the next **Put** or **Get** will be done. The **Position** parameter specifies the number of bytes from the location specified by the **SeekFlag** parameter. (See **SeekFlag** in the *Constants* section of this manual.)

```
' Set seek pointer to 100 bytes from the start
' of the stream.
ret = Stream.Seek (100, STG_STREAM_SEEK_SET)
' Now this gets the 101th byte of information
Dim bt as Byte
Dim v as Variant
v = bt ' set the variant to act like a byte
Stream.Get SEEK_DONTMOVE, v
```

Seek returns a long integer which contains the current position of the seek pointer. You can locate the current position of the seek pointer with this line:

```
CurrPos = Stream.Seek (0, STG_STREAM_SEEK_CUR)
```

dStream.SetSize (NewSize as Long)

This sets the probable maximum size of this storage. The Compound File will then allocate the appropriate amount of (usually contiguous) memory within the file. This improves the efficiency of the entire file without limiting the size of the stream. You cannot set the size of a stream smaller than the size of the data already stored within it.

```
' now the stream is 1000 bytes long
Stream.SetSize 1000
```

*dStream.**VariantGet***
(SeekPosition as Long, Buffer as Variant)

Reads information in Visual Basic Variant Binary form from this stream (that is, the way Variants are read from Binary files). The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to read at the location specified by the last **Seek** or at where the last **VariantGet** or **VariantPut** concluded. The amount of data read depends upon the size and type of the data written before, because the type information is saved in the file with the data. The Variant **Buffer** is automatically converted to the correct type. See **VariantPut** for sample code.

*dStream.**VariantGetEx***
(Long lSeekPosition, Variant pvBuffer)

This performs the same operation as VariantGet, except that a long integer (32 bytes) is used to store the length of strings instead of the short integer (16 bytes) of VariantGet. This allows the reading of strings longer than 32K in length. This method is not compatible with VariandPut - string data written by VariantPut will not be read correctly by VariantGetEx.

```
' VB
Dim v as Variant

' read it, starting at the start of the stream
dStrm.VariantGetEx 0, v
' print our very long string
Debug.Print v

' VB.NET
Dim v as Object
' read it, starting at the start of the stream
dStrm.VariantGetEx (0, v)

// C#
object v
// note the use of the "ref" keyword in C#
dStrm.VariantGetEx (0, ref v)
```

dStream.***VariantPut***
(SeekPosition as Long, Buffer as Variant)

Writes information to this stream in the way Visual Basic writes Variants to Binary data files. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to write at the location specified by the last **Seek** or at where the last **VariantGet** or **VariantPut** concluded. The amount of information written depends upon the length of the data in the variant **Buffer**. If the data is written past the end of the stream, the stream automatically expands.

```
' VB
Dim v as Variant
Dim ByteArray(10) as Byte
Dim j as Integer

' Write a string to the start of the stream
v = "abcdefghij"
Stream.VariantPut 0, v
' Write a 10-byte byte array after the string
For j = 0 to 9
    ByteArray(j) = j
Next j
v = ByteArray
Stream.VariantPut SEEK_DONTMOVE, v

' Now read the string at the start of Stream
Stream.VariantGet 0, v
' This will print "abcdefghij" and the vartype
' of a string.  Note that you do not have to
' pre-set the variant to any type - it is
' automatically set.
Debug.Print v, VarType(v)
' Now read the byte array after the string
Stream.VariantGet SEEK_DONTMOVE, v
' Print the first byte in the byte array ("0")
Debug.Print v(0)

' VB.NET
Dim v as Object
Dim ByteArray(10) as Byte
Dim j as Integer

' Write a string to the start of the stream
v = "abcdefghij"
Stream.VariantPut (0, v)
```

```
' Write a 10-byte byte array after the string
For j = 0 to 9
    ByteArray(j) = j
Next j
v = ByteArray
Stream.VariantPut (SEEK_DONTMOVE, v)

' Now read the string at the start of Stream
Stream.VariantGet (0, v)
' Now read the byte array after the string
Stream.VariantGet (dwStg.SEEK_DONTMOVE, v)

// C#
object v;
byte [] ByteArray = new byte[10];
int j;

// Write a string to the start of the stream
v = "abcdefghij";
Stream.VariantPut (0, v);
// Write a 10-byte byte array after the string
for (j = 0; j < 10; j++)
{
    ByteArray[j] = j;
}
v = ByteArray;
Stream.VariantPut (dwStg.SEEK_DONTMOVE, v);

// Now read the string at the start of Stream
Stream.VariantGet (0, ref v);
// Now read the byte array after the string
Stream.VariantGet (dwStg.SEEK_DONTMOVE, ref v);
```

*dStream.**VariantPutEx***
(Long lSeekPosition, Variant pvBuffer)

This performs the same operation as VariantPut, except that a long integer (32 bytes) is used to store the length of strings instead of the short integer (16 bytes) of VariantPut. This allows the writing of strings longer than 32K in length. This method is not compatible with VariandGet - string data written by VariantPutEx will not be read correctly by VariantGet.

```
' VB
Dim v as Variant
```

```
' create a very long string
v = String (100000, 65)
' write it, starting at the start of the stream
dStrm.VariantPutEx 0, v

' VB.NET
Dim v as Object
v = New String (" ")
v = v.PadLeft (1000000, "A"c)
dStrm.VariantPutEx (0, v)

// C#
object v;
v = new string (" ");
v = v.PadLeft (1000000, "A"c)
dStrm.VariantPutEx (0, v);
```

Which Methods Should I Use To Read And Write Data?

Get and **Put** both access information in a stream in much the same way that the Visual Basic **Get** and **Put** commands access information in a file. This is the most efficient method because only the data itself is placed into the stream. However, it means that you have to remember the type and the length of every piece of information exactly. **VariantGet** and **VariantPut** do not have this problem. These methods identify the type and length of every piece of information in the stream. This does result in some wasted space - more if you are storing a number of individual items, less if you are storing large arrays. **GetString** and **PutString** place information into the stream in the form of strings. These are provided for compatibility with the Visual Basic Sequential format, but their primary advantage (creating a mostly human readable text file) is nullified by the binary nature of Structured Storage files. **GetRecord** and **SetRecord** are also provided, but are difficult to use. (The way in which user defined types are formatted within Visual Basic prevents controls from directly accessing them.) **GetBlock** and **PutBlock** are advanced functions for those programmers dealing with large blocks of data. **GetPicture** and **PutPicture** are used for Picture properties or variables of type StdPicture only. If you are dealing with very long strings (more than 32,000 characters) then you should use VariantGetEx and VariantPutEx instead of VariantGet and VariantPut.

dStream Object Properties (dwStg.dll)

*dStream.**EOF***

This serves the same purpose as the Visual Basic EOF function - that is, it returns True if the seek pointer is at the end of the stream. You can use this to ensure that you do not read past the end of the stream (remember that writing past the end of the stream, even if you set a maximum size with **SetSize**, simply expands the stream). The following is an example of how to use this function. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

```
' VB
If (Stream.EOF) Then
    ' Stop Reading.
End If

' VB.NET
If (Convert.ToBoolean (Stream.EOF) = True) Then
   ' Stop Reading.
End If

// C#
if (Convert.ToBoolean (Stream.EOF) == true)
{
    // Stop Reading.
}
```

*dStream.**IsValid***

A Boolean value, True if the dStream represents a stream and False if not. A dStream variable is not valid until it has been set using a successful call to the CreateStream or OpenStream methods. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

*dStream.**Name***

The name of the Object. Begins as the name of the stream itself. Use the **RenameElement** method to rename the actual streams and storages.

ActiveX Storage Control Methods

Control.*AllocateMemoryHandle () as Long*

This creates a global sharable block of memory suitable for Structured Storages. The block of memory uses 0 bytes initially, but expands to accommodate any data written to the Structured Storage. You may have more than one memory handle open at a time.

```
Dim hGlobal as Long

hGlobal = DwStorage.AllocateMemoryHandle()
```

Control.*CompressStorageFile (Filename as String)*

After heavy usage, Compound Files become fragmented and can contain several gaps that take up disk space. This method unfragments and compresses the Compound File. This method is time intensive and should not be used often, certainly *not* every time the file is modified. When used on significantly compressed files it might actually make the file larger.

```
DwStorage.CompressStorageFile
        ("C:\MyDocuments\report.doc")
```

Control.*CreateStorageFile (Filename as String, AccessMode as Long) as Object*

Creates a new blank Compound File on the disk. The **Filename** parameter should contain the name (with full path, as necessary) of the file to create. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **CreateStorageFile** is successful, a dStorage object is returned that represents the root directory of the Structured Storage. If not, an error is triggered (see the *Possible Errors* section for a description of such errors).

This method can be used to open files that are not in the Structured Storage format. Include STG_CONVERT as one of the access mode flags. The file will appear to have a single stream named "Contents", which will contain the entire file contents. Simply reading from the file will cause no changes, but modifying anything will change the file into Structured Storage format permanently.

```
Dim RootStorage as Object

' Create a file temp.stg if it does not exist
' or open it if it does.  Let other programs
' access the file as well.
Set RootStorage = DwStorage.CreateStorageFile
        ("C:\temp.stg", STG_CREATE or STG_DIRECT or
        STG_READWRITE or STG_SHARE_DENYNONE)
' If the file temp.stg exists, overwrite it
' if it does not exist, create it
Set RootStorage = DwStorage.CreateStorageFile
        ("C:\temp.stg", STG_OVERWRITE or STG_DIRECT
        or STG_READWRITE or STG_SHARE_EXCLUSIVE)
```

Control.*CreateStorageMemory (Name as String, AccessMode as Long, hGlobal as Long) as Object*

Creates a new blank Structured Storage in memory at the location specified by the global memory handle **hGlobal**. (See **AllocateMemoryHandle** for information as to how to create one of these handles.) The storage is given the name found in the **Name** parameter, and is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. Memory storages are useful as a mode of organizing memory and allowing easy saving of data to disk. The Structured Storage initially utilizes no memory but expands to accommodate any data written to it.

If **CreateStorageMemory** is successful, a Storage Object is returned that represents the root directory of the Structured Storage. If not, an error is invoked. This storage can be copied to an existing storage on disk using the **CopyTo** or **MoveElementTo** methods.

```
Dim hGlobal as Long
Dim RootStgInMem as Object
Dim RootStgOnFile as Object

hGlobal = DwStorage.AllocateMemoryHandle()
Set RootStgInMem = DwStorage.CreateStorageMemory
        ("MemStg", STG_CREATE Or STG_READWRITE Or
        STG_DIRECT Or STG_SHARE_EXCLUSIVE, hGlobal)
' Now you can act on RootStgInMem just
' like any other storage.
...
' You can copy the entire contents to a file
```

```
' with a command similar to this:
RootStgInMem.CopyTo RootStgOnFile
' After you are done with the storage:
Set RootStgInMem = Nothing
DwStorage.DeallocateMemoryHandle(hGlobal)
```

Control.*DeallocateMemoryHandle (hGlobal as Long)*

This deallocates the block of memory allocated previously. Use this method after the Structured Storage is cleared (after the object is set to **Nothing**).

```
DwStorage.DeallocateMemoryHandle hGlobal
```

Control.*IsStorageFile (Filename as String) as Boolean*

Returns True if the file specified by the **Filename** parameter is a Compound File, False if it is not. If something goes wrong, an error is generated. Remember that non-Compound Files can still be opened as Compound Files. (This is done by using CreateStorageFile, specifying the file to be converted, and setting the STG_CONVERT flag. See the STG_CONVERT flag in the *Constants* section of this manual for more information.)

```
If (DwStorage.IsStorageFile ("C:\report.doc")) Then
    MsgBox "Yes, report.doc is a storage file"
EndIf
```

Control.*IsStorageMemory (hGlobal as Long) as Boolean*

Returns True if the memory specified by the global memory handle **hGlobal** contains a Structured Storage, False if not.

```
If (DwStorage.IsStorageMemory (hGlobal)) Then
    MsgBox "Yes, the memory referenced by handle
  hGlobal does contain a storage"
EndIf
```

Control.***OpenStorageFile (Filename as String, AccessMode as Long) as Object***

Opens a Compound File on disk. The **Filename** parameter contains the name (with full path, if needed) of the file to open. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **OpenStorageFile** is successful, a Storage Object is returned that represents the root directory of the Structured Storage. If not, an error is generated.

```
Dim RootStorage as Object

' Open the file temp.stg for reading and
' writing and do not let anyone else access
' it.
Set RootStorage = DwStorage.OpenStorageFile
        ("C:\temp.stg", STG_DIRECT or STG_READWRITE
        or STG_SHARE_EXCLUSIVE)
' Open the file temp.stg for reading only.
' Let anyone else read and write to the file.
Set RootStorage = DwStorage.OpenStorageFile
        ("C:\temp.stg", STG_DIRECT or STG_READ or
        STG_SHARE_DENYNONE)
```

Control.***OpenStorageMemory (Name as String, AccessMode as Long, hGlobal as Long) as Object***

Opens an already existing Structured Storage at the location in memory specified by the global memory handle **hGlobal**. It is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. The Structured Storage expands to accommodate any data written to it. If **OpenStorageMemory** is successful, a Storage Object is returned that represents the root directory of the Structured Storage. This storage can be copied to disk using the **CopyTo** or **MoveElementTo** methods.

```
Dim RootStgInMem as Object

' In this case, we probably get hGlobal from
' another task or module.
```

```
Set RootStgInMem = DwStorage.OpenStorageMemory
       ("MemStg", STG_READWRITE Or STG_DIRECT Or
       STG_SHARE_EXCLUSIVE, hGlobal)
' Now you can act on RootStgInMem just
' like any other storage.
...
' After you are done with the storage:
Set RootStgInMem = Nothing
```

ActiveX Storage Object Methods

*StorageObject.**Commit***
(CommitFlags as Long)

If the storage is opened in **Transacted** mode (see Access Flags in the *Constants* section of this manual), this procedure makes permanent all changes made to the storage since it was opened or since the last **Commit**. It makes reverting any changes up to this point impossible. This call is also needed if the storage is opened in **Direct** mode to flush the buffer. **CommitFlags** describes how the storage should be committed (see the *Constants* section of this manual).

```
Dim RootStg as dStorage
' this performs a normal commit
RootStg.Commit 0
```

*StorageObject.**CopyTo***
(DestinationStorage as Object)

Copies the contents of this storage to the already opened Storage Object passed to the **DestinationStorage** parameter.

```
Dim RootStg as dStorage
Dim backupStg as dStorage
' Copy the entire contents of one Storage file
' into the backupStg storage (which may be in
' another file or in memory).
RootStg.CopyTo backupStg
```

*StorageObject.**CreateStorage***
(StorageName as String, AccessMode as Long) as Object

Creates a sub-Storage in this Storage. The **Filename** parameter contains the name of the storage to be constructed. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **CreateStorage** is successful, a Storage Object is returned that represents the newly created storage. If not, an error is generated.

```
Dim RootStg as Object
Dim newStorage as Object
' Create a storage named DataBlocks in the
' RootStg storage if one does not exist, or
```

```
' open the storage named DataBlocks if it
' does.  This storage (like all non-root
' storages under OLE Structured Storage must
' be) cannot be accessed by other applications
' while the newStorage object exists.
Set newStorage = RootStg.CreateStorage
       ("DataBlocks", STG_CREATE or STG_READWRITE
       or STG_SHARE_EXCLUSIVE)
' When you are done with the storage, get rid
' of the object.
Set newStorage = Nothing
```

StorageObject.***CreateStream (StreamName as String, AccessMode as Long) as Object***

Creates a sub-stream in this storage. The **Filename** parameter contains the name of the stream to be constructed. The file is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **CreateStream** is successful, a Stream Object is returned that represents the newly created stream. If not, an error is generated.

```
Dim RootStg as Object
Dim subStream as Object
' Create a stream named Data1 in the RootStg
' storage if one does not exist, or open the
' stream named Data1 if it does.  This stream
' (like all streams under OLE Structured
' Storage must be) cannot be accessed by other
' applications while the newStream object
' exists.
Set subStream = RootStg.CreateStream ("Data1",
       STG_CREATE or STG_READWRITE or
       STG_SHARE_EXCLUSIVE)
' When you are done with the stream, get rid
' of the object.
Set newStream = Nothing
```

StorageObject.***DestroyElement (Name as String)***

Deletes the named storage or stream from this storage. If the element being deleted is a storage, then all the elements in that storage are also deleted. If there is any problem, an error is sent.

```
' Deletes the stream or storage named Data1
' in the RootStg storage.
RootStg.DestroyElement "Data1"
```

StorageObject.***Directory (Index as Long, Type as Integer) as String***

This method can be used to obtain a catalogue of all storages and streams contained in this storage. Each call to **Directory** returns the name of the storage or stream specified by the **Index**. The **Type** integer is changed to reveal the type of the element: **STG_TYPE_STREAM** if the returned name represents a stream, **STG_TYPE_STORAGE** if the returned name represents a storage, and **STG_TYPE_NONE** if there are no more elements.

```
' enumerate through all the elements in the
' RootStg storage.  After the loop is done, the
' variable i will contain the count of elements
i = 0
Do
      ElementName = RootStg.Directory(i, FileType)
      If FileType = STG_TYPE_NONE Then Exit Do
      If FileType = STG_TYPE_STORAGE Then
          ' ElementName is a storage
      ElseIf FileType = STG_TYPE_STREAM Then
          ' ElementName is a stream
      End If
      i = i + 1
 Loop
```

StorageObject.***EnumDirectory (RelativePos as Long, Type as Integer) as String***

This method can be used to obtain a catalogue of all storages and streams contained in this storage. It is faster than the normal **Directory** method, but is less flexible. Each call to **EnumDirectory** returns the name of the first storage or stream if **RelativePos** is equal to 0, or the next storage or stream if **RelativePos** is not. The **Type** integer is changed to reveal the type of the element: **STG_TYPE_STREAM** if the returned name represents a stream, **STG_TYPE_STORAGE** if the returned name represents a storage, and **STG_TYPE_NONE** if there are no more elements.

```
' enumerate through all the elements in the
' RootStg storage.  After the loop is done, the
```

```
' variable i will contain the count of elements
i = 0
ElementName = RootStg.EnumDirectory(0, FileType)
Do
      If FileType = STG_TYPE_NONE Then Exit Do
      If FileType = STG_TYPE_STORAGE Then
          ' ElementName is a storage
      ElseIf FileType = STG_TYPE_STREAM Then
          ' ElementName is a stream
      End If
      ElementName = RootStg.EnumDirectory(1,
         FileType)
      i = i + 1
Loop
```

*StorageObject.**GetCreationDate () as Date***

Obtains the date this storage was created.

```
Dim dt as Date
dt = RootStg.GetCreationDate
```

*StorageObject.**GetLastModifyDate () as Date***

Obtains the date this storage was last changed in any way.

```
Dim dt as Date
dt = RootStg.GetLastModifyDate
```

*StorageObject.**GetLastAccessDate () as Date***

Obtains the date this storage was last read. Not all file systems carry this information -- in which case **GetLastAccessDate** will return a zero as the last date of access.

```
Dim dt as Date
dt = RootStg.GetLastAccessDate
If (dt = CDate(0)) Then
    MsgBox "Does not support Last Access time"
Else
    MsgBox "Last Access: " & CDate (dt)
EndIf
```

StorageObject.***MoveElementTo (ElementName as String, NewName as String, DestStorage as Object, MoveFlags as Long)***

Copies or moves an element from within this storage to the already open storage object passed to the **DestStorage** parameter. The element can be renamed by putting a different name in **NewName**. If **NewName** is **Null**, then the old name is kept. If **MoveFlags** is **STG_MOVEMOVE** then the original is erased (a move); if **MoveFlags** is **STG_MOVECOPY** then the original is not erased (a copy).

```
' Copy the stream Data1 in the RootStg storage
' into the storage named DataBackup.  Keep the
' same filename.
RootStg.MoveElementTo "Data1", vbNullString,
     DataBackup, STG_MOVECOPY
' Move the stream Data1 in the RootStg storage
' into the storage named DataBackup.  Change
' the filename to Yesterday1.
RootStg.MoveElementTo "Data1", "Yesterday1",
     DataBackup, STG_MOVEMOVE
```

StorageObject.***OpenStorage (StorageName as String, AccessMode as Long) as Object***

Opens a storage that is in the same file as this storage. The **Filename** parameter contains the name (and path, as necessary) of the storage to open. Like changing directories in DOS, the path can be relative to the root (with a leading backslash ("\")) or relative to this storage (without the leading backslash). The storage is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **OpenStorage** is successful, a Storage OLE Object is returned that represents the newly opened storage. If not, an error is generated.

```
Dim RootStg as Object
Dim subStorage as Object
' Open a storage named DataBlocks in the
' RootStg storage.  This storage (like all
' non-root storages under OLE Structured
' Storage must be) cannot be accessed by other
' applications while the subStorage object
' exists.
```

```
Set subStorage = RootStg.OpenStorage ("DataBlocks",
        STG_DIRECT or STG_READWRITE or
        STG_SHARE_EXCLUSIVE)
' When you are done with the storage, get rid
' of the object.
Set subStorage = Nothing
```

StorageObject.***OpenStream (StreamName as String, AccessMode as Long) as Object***

Opens a sub-stream of this storage. The **Filename** parameter contains the name of the stream to open. The stream is opened in the style dictated by the flags in the **AccessMode** parameter. The possible flags are detailed in the *Constants* section of this manual. If **OpenStream** is successful, a Stream OLE Object is returned that represents the newly opened stream. If not, an error is generated.

```
Dim RootStg as Object
Dim subStream as Object
' Create a stream named Data1 in the RootStg
' storage if one does not exist, or open the
' stream named Data1 if it does.  This stream
' (like all streams under OLE Structured
' Storage must be) cannot be accessed by other
' applications while the newStream object
' exists.
Set subStream = RootStg.OpenStream ("Data1",
        STG_READWRITE or STG_SHARE_EXCLUSIVE)
' When you are done with the stream, get rid
' of the object.
Set subStream = Nothing
```

StorageObject.***RenameElement (OldName as String, NewName as String)***

Renames one of the storages or streams in this storage. Compound File names are limited to thirty-two characters in length.

```
' This renames the element named Data1 in
' storage RootStg to ReserveData1
 RootStg.RenameElement "Data1", "ReserveData1"
```

*StorageObject.**Revert ()***

When this method is called, if the storage is opened in **Transacted** mode, then all changes made to the storage since it was created or since the last **Commit** are discarded. **Revert** has no effect in **Direct** mode.

```
Dim RootStg as Object
Dim subStream as Object

' Open an existing file in Transacted mode.
Set RootStg = dwStorage1.OpenStorageFile
  ("C:\file.stg", STG_TRANSACTED or STG_READWRITE
  or STG_SHARE_EXCLUSIVE)
' Create a new stream.
Set subStream = RootStg.CreateStream ("Data1",
  STG_CREATE or STG_DIRECT or STG_READWRITE or
  STG_SHARE_EXCLUSIVE)
' Now write the stream.  If the RootStg was
' Commited, it would contain a blank stream
' named "Data1".
subStream.Flush 0
Set subStream = Nothing
' This undoes all the changes I made to the
' RootStg storage - the stream subStream does
' not get written to the file.
RootStg.Revert
```

*StorageObject.**SetElementTimes***
(Element as String, CreationDate as Date, LastModifyDate as Date, LastAccessDate as Date)

Sets any or all of the dates for a sub-storage of the storage named in **Element** (streams do not carry any date information in the current implementation of OLE2). Use **Null** for any date you do not want changed. **LastAccessDate** is not saved on FAT or NTFS file systems, so changing it might not be useful.

```
Dim Date1 as Date
Dim Date2 as Date

Date1 = Now
Date2 = Now
' Set the creation and last modify dates for
' the storage named "DataBlock1" in storage
' RootStg to the current date and time.  I only
' pass a zero for the last access time.
```

```
RootStg.SetElementTimes "DataBlock1", Date1, Date2,
        CDate(0)
```

Summary Information Property Set Functions

These functions deal with the Summary Information stream in the relevant Storage object.

The following is an example of how you would use these methods to edit a string in the Summary Information Property set.

```
' if the SummaryInformation stream exists, load
' its information
If (RootStorage.siOpenSummaryInfo = True) Then
    ' get the title
    titleString = RootStorage.siGetTitle()
    titleString = titleString & " version 2"
    ' set the new title
    RootStorage.siSetTitle titleString
    ' save the changes to the
    ' SummaryInformation stream
    RootStorage.siSaveSummaryInfo
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
     RootStorage.Commit STG_DEFAULT
 End If
```

StorageObject**.siSetTitle (Title as String)**

Sets the title of the document.

StorageObject**.siGetTitle () as String**

Returns the title of the document.

StorageObject**.siSetSubject (Subject as String)**

Sets the subject of the document.

StorageObject**.siGetSubject () as String**

Returns the subject of the document.

StorageObject.siSetAuthor (Author as String)

Sets the author of the document.

StorageObject.siGetAuthor () as String

Returns the author of the document.

StorageObject.siSetKeywords (Keyword as String)

Sets key words related to the subject of the document. These keywords are typically used in search routines.

StorageObject.siGetKeywords () as String

Returns keywords relating to the subject of the document.

StorageObject.siSetComments (Comment as String)

Sets the comment field. This area might be used for making notes to oneself or others.

StorageObject.siGetComments () as String

Returns the comment field.

StorageObject.siSetLastAuthor (LastAuthor as String)

Sets the last person to have modified the document.

StorageObject.siGetLastAuthor () as String

Returns the last person who has modified the document.

StorageObject.siIncrementRevNum ()

Increments the number of times a document has been revised.

StorageObject.siSetRevNum (TimesRevised as Long)

Sets the number of times the document has been revised.

StorageObject.siGetRevNum () as Long

Returns the number of times the document has been revised.

StorageObject.siStartEditTimer ()

This starts an internal timer. It is useful for determining how long a document has been opened.

StorageObject.siAddEditTimerToTotal ()

Adds the amount of time since **siStartEditTimer** was called to the field recording the total amount of time the document has been opened.

StorageObject.siSetTotalEditTime (Minutes as Long)

Sets the total amount of time a document has been open, in minutes.

StorageObject.siGetTotalEditTime () as Long

Returns the total amount of time a document has been open since it was created, in minutes.

StorageObject.siRecordPrintDate ()

Sets the field containing the last time this document was printed to the current date and time.

StorageObject.siGetLastPrintDate () as Date

Returns the last time the document was printed.

StorageObject.siRecordCreateDate ()

Sets the field containing the time the document was created to the current date and time.

StorageObject.siGetCreateDate () as Date

Returns the time the document was created.

StorageObject.siRecordSaveDate ()

Sets the field containing the last time this document was saved to the current date and time.

StorageObject.siGetLastSaveDate () as Date

Returns the last time the document was saved.

StorageObject.siSetNumberOfPages (NumPages as Long)

Sets the number of pages.

StorageObject.siGetNumberOfPages () as Long

Returns the number of pages.

StorageObject.siSetNumberOfWords (NumWords as Long)

Sets the number of words.

StorageObject.siGetNumberOfWords () as Long

Returns the number of words.

StorageObject.siSetNumberOfCharacters (NumChars as Long)

Sets the number of characters.

StorageObject.siGetNumberOfCharacters () as Long

Returns the number of characters.

StorageObject.siSetApplication (AppName as String)

Sets the name of the application that created the document.

StorageObject.siGetApplication () as Long

Returns the title of the application that created the document.

StorageObject.siSetTemplate (Template as String)

Sets the filename of the template used in the document.

StorageObject.siGetTemplate () as String

Returns the filename of the template used in the document.

StorageObject.siSetSecurity (SecurityLevel as Long)

Sets the recommended security level of the document. Note that this does not actually supply any security, but only serves as a reminder to the application that reads the file. The actions based on the values are as follows:

Value	Security Procedure
0	None.
1	Password protected. Do not allow viewing or editing of this document. Other programs cannot view or edit this document.
2	Read-Only recommend. The user will be warned if there is any attempt to edit the document.
3	Read-Only enforced. Does not allow any changes to the document.
4	Locked for Annotations. Does not allow any changes to the document.

StorageObject.siGetSecurity () as Long

Returns the security level of the document. Your program should behave as follows:

Value	Security Procedure
0	None.
1	Password protected. Do not allow viewing or editing of this document unless you know what type of password protection is involved, and the password given is correct.
2	Read-Only recommend. Warn the user if there is any attempt to edit the document.
3	Read-Only enforced. Do not allow any changes to the document.
4	Locked for Annotations. Do not allow any changes to the document.

StorageObject.siOpenSummaryInfo () as Boolean

Retrieves information from the SummaryInformation Property Set stream in the root storage of this Compound File. If there is no SummaryInformation Property Set, **siOpenSummaryInfo()** generates an error. Before the **siOpenSummaryInfo()** is called, using any function beginning with "siGet" will return the default value, which will be either zero or a blank string depending upon the type.

StorageObject.siSaveSummaryInfo ()

Saves any changes to the SummaryInformation Property Set stream in the root storage. Just as with any stream, these changes are not actually written to disk until the root storage object's **Commit** method is called.

```
' Create a new storage and create a new SI
 Dim dwStg as Object

 Set dwStg = RootStorage.CreateStorage ("siTest",
   STG_CREATE or STG_READWRITE Or STG_DIRECT Or
   STG_SHARE_EXCLUSIVE)

 dwStg.dsiSetNumWords 412
 ' You do not have to set all properties – when you
 ' save, all the rest of the properties will be
 ' saved with their default values.
 dwStg.dsiSaveSummaryInfo
 dwStg.Commit 0
 Set dwStg = Nothing
```

Document Summary Information Property Set Functions

These functions deal with the Document Summary Information stream in the relevant Storage object. Currently, the ActiveX control cannot write to the Document Summary Information stream, nor can it access the user-defined properties.

Here is an example of how you would use these methods to read a string in the Document Summary Information Property set:

```
' if the DocumentSummaryInformation stream
' exists, load its information
If (RootStorage.dsiOpenSummaryInfo = True) Then
     ' get the name of the manager
    mngString = RootStorage.dsiGetManager()
```

```
End If
```

StorageObject.dsiGetScaleCrop () as Boolean

Returns True if this document is supposed to be scaled to the current dimensions of the window, False if it is to be cropped to the dimensions of the window.

StorageObject.dsiGetLinksUpToDate () as Boolean

Returns True if all the links are up to date.

StorageObject.dsiGetCategory () as String

Returns the category of the document.

StorageObject.dsiGetPresentationTarget as String

Returns the presentation target of the document.

StorageObject.dsiGetManager () as String

Returns the manager of the document writer.

StorageObject.dsiGetCompany () as String

Returns the name of the company with which the document is associated.

StorageObject.dsiGetNumBytes () as Long

Returns the size of the document in number of bytes.

StorageObject.dsiGetNumLines () as Long

Returns the number of lines in the document.

StorageObject.dsiGetNumParagraphs () as Long

Returns the number of paragraphs in the document.

StorageObject.dsiGetNumSlides () as Long

Returns the number of slides in the document.

StorageObject.dsiGetNumNotes () as Long

Returns the number of notes in the document.

StorageObject.dsiGetNumHiddenSlides () as Long

Returns the number of hidden slides in the document.

StorageObject.dsiGetNumMMClips () as Long

Returns the number of multimedia clips in the document.

StorageObject.dsiOpenDocSummaryInfo () as Boolean

Retrieves information from the Document Summary Information Property Set stream. If there is no Document Summary Information Property Set, **dsiOpenDocSummaryInfo** returns False. Before the **dsiOpenDocSummaryInfo** method is called, or if there is no Document Summary Information stream, using any "dsi" method beginning with "Get" will return the default value, which will be either zero or a blank string depending on the type. To make a new Document Summary Information stream where one does not yet exist, set the above properties to the values you want and then call the **dsiSaveDocSummaryInfo** method.

ActiveX Storage Object Properties

*StorageObject.**Name***

The name of the storage object. Begins as the name of the storage itself. For root storages, the storage name is the same as the filename. Use the **RenameElement** method to rename the actual streams and storages, or Windows file system methods to rename the file.

```
' If I have the object, I might still need the
' name for some function calls.
 RootStg.RenameElement subStg.Name, "NewName"
```

ActiveX Stream Object Methods

*StreamObject.**Flush***
(CommitFlags as Long)

In the current implementation of OLE2, streams can only be opened in Direct mode. **Flush** flushes any internal buffers (although this only speeds up what would have been done by a few seconds in any case). **CommitFlags** illustrates how the stream should be committed. (See the *Constants* section of this manual.)

```
' Perform a normal flush of buffers
Stream.Flush 0
```

*StreamObject.**Get***
(SeekPosition as Long, Buffer as Variant)

Reads information in Visual Basic binary form from this stream. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to read at the location specified by the last Seek or at where the last **Get** or **Put** operation concluded. The amount of data read depends upon the size and type of the variant **Buffer.**

```
' Read in two 10-character strings from the start
' of the stream (the first 20 bytes). No matter what
' the data originally meant, Get will interpret it
' as the type of the Variant passed to it.
Dim Buffer as Variant
' Pre-set the variant to 10 characters
Buffer = String (10, 0)
Stream.Get 0, Buffer
Debug.Print Buffer
' Now get the second 10 character string
Stream.Get SEEK_DONTMOVE, Buffer
Debug.Print Buffer
```

StreamObject.***GetBlock***
(SeekPosition as Long, NumOfBytes as Long) as Long

GetBlock can be used to get a large block of data from a stream. This may be useful if you are trying to manipulate data in a stream larger than the limits of what a string or array can contain. It returns a pointer to an internally allocated block holding the data from the stream starting at **SeekPosition**. If there is any problem in allocating the block, the return value is 0. If **NumOfBytes** is larger than the number of bytes in the stream, the buffer from the end of the data onward is filled with zeroes.

The internal buffer is automatically deleted when the stream object is destroyed (for example, when it is set to Nothing or set to another stream), so be sure to keep the stream object as long as you are accessing the memory block. Using this method will release any previously allocated buffer.

```
' Points to the block of memory where I load
' the data from the stream.
Dim BlockPtr As Long
' Byte array where I will copy the data.  This
' will make it easier to manage in Visual Basic
Dim datablock() As Byte

' Get all the data in the stream.
BlockPtr = stream.GetBlock(0, stream.GetSize())
' Adjust the size of the byte array to that of
' the block I just read.
ReDim datablock(stream.GetSize())

' Copy the data from the pointer to the byte
' array.  The "stg*" functions are part of the
' dwAddr16.dll and dwAddr32.dll support libraries.
stgCopyDataBynum BlockPtr,
   stgGetAddressForObject(datablock(0)),
   stream.GetSize()

' Now all the data in the stream is in the
' datablock byte array.  After you are done
' with the data, write it back:
 stream.PutBlock 0, stream.GetSize(),
        stgGetAddressForObject(datablock(0))
```

*StreamObject.**GetPicture () as Picture***

GetPicture is used to read a picture from the stream. See **PutPicture** for information on how the picture is stored into a stream. **GetPicture** is compatible with the Picture property of forms, picture boxes, image boxes and other controls.

```
' Get the picture in stream PicStream
' Picture1 is a picture box control
Picture1.Picture = PicStream.GetPicture ()
```

*StreamObject.**GetRecord (RecordNumber as Long, RecordSize as Long, RecordAddress as Long)***

(Currently available in 16 bit Windows only. See DWADDR32.dll for 32 bit Windows.)

Reads a block of information from the record **RecordNumber** of size **RecordSize**. The **RecordNumber** parameter specifies the number of records from the start of the file, beginning with zero. All records in the stream must be of the same size and must match the size of the record exactly in order to prevent corruption of data. Use the constant **SEEK_DONTMOVE** to read at the location specified by the last **Seek** or where the last **GetRecord** or **PutRecord** terminated. If you do use **Seek**, be sure to seek to a whole multiple of the size of the record. You can get the address of a record by using the **dwGetAddressForRecord** function from the DWADDR16.DLL.

*StreamObject.**GetSize() as Long***

Returns the size, in bytes, of this stream.

```
' Get the size in number of bytes of a stream
 NumBytes = PicStream.GetSize
```

*StreamObject.**GetString (StringData as String)***

Obtains a string from the Visual Basic Sequential formatted stream. The string returned is changed to the correct size. You will need to convert the string to the correct type using Visual Basic conversion functions. The seek pointer should be pointing to the start of the item to retrieve. After **GetString** is complete, the seek pointer will point to the next item.

```
' If the next data item in the stream is the
' integer “123”, the statement
Stream.GetString InputString
' will return the string “123”, which you can
' convert to an integer with:
InputInteger = CInt(InputString)
```

StreamObject.***Put (SeekPosition as Long, Buffer as Variant)***

Writes information to this stream in the same way Visual Basic writes Binary data. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to write at the location specified by the last **Seek** or at where the last **Get** or **Put** concluded. The amount of information written depends upon the length of the data in the variant **Buffer**. If the data is written past the end of the stream, the stream automatically expands.

```
' Save a 10-character string at the start of
' the stream.
Dim Buffer as Variant
Buffer = "abcdefghij"
Stream.Put 0, Buffer
' Save a long integer to the place in the
' stream where the seek pointer currently is.
Buffer = 123456
Stream.Put SEEK_DONTMOVE, Buffer
```

StreamObject.***PutBlock (SeekPosition as Long, NumOfBytes as Long, Pointer as Long)***

PutBlock is used to write a large block of data to a stream. This may be useful if you are trying to manipulate data larger than the limits of what a string or array can contain. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to write to the location specified by the last **Seek** or at where the last stream method concluded. **NumOfBytes** specifies the number of bytes, starting at memory location **Pointer**, that are written to the stream. The dwAddress .DLL’s come with functions for obtaining pointers and manipulating data within the block.

See GetBlock for a sample.

*StreamObject.**PutPicture (Pic as Picture)***

PutPicture is used to write a picture to the stream. The picture will take up the entire stream, and it will not be possible to write other information to the stream without corrupting the picture. This is compatible with the Picture property of forms, picture boxes, image boxes and other controls.

```
' Picture1 is a picture box control
PicStream.PutPicture Picture1.Picture
```

*StreamObject.**PutRecord (RecordNumber as Long, RecordSize as Long, RecordAddress as Long)***

(Currently available in 16 bit Windows only.)

Writes a block of information to the record **RecordNumber** of size **RecordSize**. The **RecordNumber** parameter specifies the number of records from the start of the file, beginning with zero. All records in the stream must be the same size and must match the size of the record exactly in order to prevent the corruption of data. Use the constant **SEEK_DONTMOVE** to write at the location specified by the last **Seek** or at where the last **GetRecord** or **PutRecord** concluded. If you do use **Seek**, be sure to seek to a whole multiple of the size of the record. You can obtain the address of a record by using the **GetAddressForRecord** function from the DWADDR16.DLL.

*StreamObject.**PutString (StringData as String, VarType as Integer, Comma as Integer)***

This writes a string to the stream in Visual Basic Sequential format. **VarType** should contain the type of the variable (obtained from Visual Basic with the VarType() function). If **Comma** is True, then commas are used to separate items; if not, then a Line Feed/Carriage Return is used. The stream automatically expands if the data is written past the end of the file.

```
' Write a string to the stream, with a comma
' afterwards
Dim StrData as String
StrData = "first string"
```

```
Stream.PutString StrData, VarType(StrData), -1
' Write a number without a comma
StrData = CStr(123)
Stream.PutString StrData, VarType(123), 0
```

StreamObject.***Seek***
(Position as Long, SeekFlag as Long) as Long

Sets the position at which the next **Put** or **Get** will be done. The **Position** parameter specifies the number of bytes from the location specified by the **SeekFlag** parameter. (See **SeekFlag** in the *Constants* section of this manual.)

```
' Set seek pointer to 100 bytes from the start
' of the stream.
ret = Stream.Seek (100, STG_STREAM_SEEK_SET)
' Now this gets the 101th byte of information
Dim bt as Byte
Dim v as Variant
v = bt ' set the variant to act like a byte
 Stream.Get SEEK_DONTMOVE, v
```

Seek returns a long integer which contains the current position of the seek pointer. You can locate the current position of the seek pointer with this line:

```
 CurrPos = Stream.Seek (0, STG_STREAM_SEEK_CUR)
```

StreamObject.***SetSize***
(NewSize as Long)

This sets the probable maximum size of this storage. The Compound File will then allocate the appropriate amount of (usually contiguous) memory within the file. This improves the efficiency of the entire file without limiting the size of the stream. You cannot set the size of a stream smaller than the data already stored within it.

```
' now the stream is 1000 bytes long
 Stream.SetSize 1000
```

*StreamObject.**VariantGet***
(SeekPosition as Long, Buffer as Variant)

Reads information in Visual Basic Variant Binary form from this stream (that is, the way variants are read from binary files). The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to read at the location specified by the last **Seek** or at where the last **VariantGet** or **VariantPut** concluded. The amount of data read depends upon the size and type of the data written before, because the type information is saved in the file with the data. The variant **Buffer** is automatically converted to the correct type.

*StreamObject.**VariantPut***
(SeekPosition as Long, Buffer as Variant)

Writes information to this stream in the way Visual Basic writes variants to binary data files. The **SeekPosition** parameter specifies the position in number of bytes from the start of the file, beginning with zero. Use the constant **SEEK_DONTMOVE** to write at the location specified by the last **Seek** or at where the last **VariantGet** or **VariantPut** concluded. The amount of information written depends upon the length of the data in the variant **Buffer**. If the data is written past the end of the stream, the stream automatically expands.

```
Dim v as Variant
Dim ByteArray as Byte(10)
Dim j as Integer

' Write a string to the start of the stream
v = "abcdefghij"
Stream.VariantPut 0, v
' Write a 10-byte byte array after the string
For j = 0 to 9
    ByteArray(j) = j
Next j
v = ByteArray
Stream.VariantPut SEEK_DONTMOVE, v

' Now read the string at the start of Stream
Stream.VariantGet 0, v
' This will print "abcdefghij" and the vartype
' of a string.  Note that you do not have to
' pre-set the variant to any type - it is
' automatically set.
```

```
Debug.Print v, VarType(v)
' Now read the byte array after the string
Stream.VariantGet SEEK_DONTMOVE, v
' Print the first byte in the byte array ("0")
Debug.Print v(0)
```

Which Methods Should I Use To Read And Write Data?

Get and **Put** both access information in a stream in much the same way that the Visual Basic **Get** and **Put** commands access information in a file. This is the most efficient method because only the data itself is placed into the stream. However, it means that you have to remember the type and the length of every piece of information exactly. **VariantGet** and **VariantPut** do not have this problem. These methods identify the type and length of every piece of information in the stream. This does result in some wasted space - more if you are storing many individual items, less if you are storing large arrays. **GetString** and **PutString** place information into the stream in the form of strings. These are provided for compatibility with the Visual Basic Sequential format, but their primary advantage (creating a mostly human readable text file) is nullified by the binary nature of Structured Storage files. **GetRecord** and **SetRecord** are also provided but are difficult to use. (The way in which user defined types are formatted within Visual Basic prevents controls from directly accessing them.) **GetBlock** and **PutBlock** are advanced functions for those programmers dealing with large blocks of data. **GetPicture** and **PutPicture** are used for Picture properties or variables of type StdPicture only.

ActiveX Stream Object Properties

*StreamObject.**EOF***

This serves the same purpose as the Visual Basic EOF function - that is, it returns True if the seek pointer is at the end of the stream. You can use this to ensure that you do not read past the end of the stream. (Remember that writing past the end of the stream, even if you set a maximum size with **SetSize**, simply expands the stream). The following is an example of how to use this function.

```
Dim IsAtEndOfStream as Boolean

IsAtEndOfStream = Stream.EOF
If (IsAtEndOfStream) Then
   '  Do nothing.
Else
   '  I can read more information.
End If
```

*StreamObject.**Name***

The name of the stream object. Begins as the name of the stream itself. Use the **RenameElement** method to rename the actual streams and storages.

Functions in the DWADDR16.DLL

(16 bit Windows only)

dwGetAddressForRecord

Declare Function dwGetAddressForRecord& Lib "dwaddr16.dll" (Record as Any). Returns the address of the specified record as a long integer. These addresses often change, so it should only be used right after this call, and not kept for any length of time. Use with **GetRecord** and **PutRecord**. When using this to get pointers to user defined types, try getting the address of the first element in the record:

```
Private Type x
    a as integer
    b as date
End Type

Dim RecordX as x
ValidAddr = dwGetAddressForRecord (RecordX.a)
```

dwGetBytesFromRecord

Declare Sub dwGetBytesFromRecord Lib "dwaddr16.dll" (ByVal RecordSize As Integer, Record As Any, Bytes As Variant)

Copies the information in the user-defined type **Record** into the array of bytes held in the variant **Bytes**. **Bytes** need not be defined. **RecordSize** is the length, in bytes, of the entire user-defined type. You can then use **Put** or **VariantPut** to save the array to a stream.

dwGetRecordFromBytes

Declare Sub dwGetRecordFromBytes Lib "dwaddr16.dll" (ByVal RecordSize As Integer, Record As Any, Bytes As Variant)

Copies the information held in the array of bytes in the variant **Bytes** into the user defined type record. **RecordSize** is the length in bytes of the entire user defined type. Verify that the **Bytes** variant array contains as many bytes as are needed to fill the record structure. In order to use this function to read a record from a stream, first read the required number of bytes into an array of bytes using **Get** or **VariantGet**. Then you can use this function to populate the record properly.

stgCopyData, stgCopyDataBynum

Private Declare Sub stgCopyData Lib "dwaddr16.dll"(ByVal source As Any, ByVal dest As Any, ByVal NumBytes As Any)

Private Declare Sub stgCopyDataBynum Lib "dwaddr16.dll" Alias "stgCopyData" (ByVal source&, ByVal dest&, ByVal NumBytes%)

These subroutines allow you to copy blocks of memory from one place to another. Use the "Bynum" version with the stgGetAddressForObject function for a higher degree of safety.

stgGetAddressForObject

Declare Function stgGetAddressForObject & Lib "dwaddr16.dll" alias "dwGetAddressForRecord" (Object as Any)

Returns the address of the specified variable as a long integer. These addresses often change, so it should only be used right after this call, and not kept for any length of time. Although you can get pointers to many types of Visual Basic variables, not all the pointers will actually be meaningful. For example, pointers to strings may not actually point to the place where the string is located. When using this to get pointers to arrays, try getting the address of the first element in the array:

```
Dim ary(20) as long
ValidAddr = stgGetAddressForObject (ary(0))
```

Functions in the DWADDR32.DLL

(32 bit Windows only)

dwGetBytesFromRecord

Declare Sub dwGetBytesFromRecord Lib "dwaddr32.dll" (ByVal RecordSize As Integer, Record As Any, Bytes As Variant)Copies the information in the user-defined type **Record** into the array of bytes held in the variant **Bytes**. **Bytes** need not be defined. **RecordSize** is the length, in bytes, of the entire user-defined type. You can then use **Put** or **VariantPut** to save the array to a stream.

dwGetRecordFromBytes

Declare Sub dwGetRecordFromBytes Lib "dwaddr32.dll" (ByVal RecordSize As Integer, Record As Any, Bytes As Variant)

Copies the information held in the array of bytes in the variant **Bytes** into the user defined type record. **RecordSize** is the length in bytes of the entire user defined type. Verify that the **Bytes** variant array contains as many bytes as are needed to fill the record structure. In order to use this function to read a record from a stream, first read the required number of bytes into an array of bytes using **Get** or **VariantGet**. Then you can use this function to populate the record properly.

stgCopyData, stgCopyDataBynum

Private Declare Sub stgCopyData Lib "dwaddr32.dll" (ByVal source As Any, ByVal dest As Any, ByVal NumBytes As Any)

Private Declare Sub stgCopyDataBynum Lib "dwaddr32.dll" Alias "stgCopyData" (ByVal source&, ByVal dest&, ByVal NumBytes%)

These subroutines allow you to copy blocks of memory from one place to another. Use the "Bynum" version with the stgGetAddressForObject function for a higher degree of safety.

stgGetAddressForObject

Declare Function stgGetAddressForObject & Lib "dwaddr32.dll" alias "dwGetAddressForRecord" (Object as Any)

Returns the address of the specified variable as a long integer. These addresses often change, so it should only be used right after this call, and not kept for any length of time. Although you can get pointers to many types of Visual Basic variables, not all the pointers will actually be meaningful. For example, pointers to strings may not actually point to the place where the string is located. When using this to get pointers to arrays, try getting the address of the first element in the array:

```
Dim ary(20) as long
ValidAddr = stgGetAddressForObject (ary(0))
```

Constants

Access Flags

Use these with **CreateStorageFile**, **OpenStorageFile**, **CreateStorageMemory**, **OpenStorageMemory**, **CreateStorage**, **OpenStorage**, **CreateStream**, or **OpenStream**.

any one of:

STG_READ = 0

Read from, but not write to the element. **Put** and other routines that write information will still work, but any attempt to **Flush** or **Commit** will result in an error. With STG_DIRECT set, this is faster and takes less memory and disk space than almost any other way of opening a file.

STG_WRITE = 1

Write to, but not read from the element.

STG_READWRITE = 2

Allows both reading and writing to the element.

any one of:

STG_SHARE_DENY_READ = 30 (hex)

Does not allow any other to read from this element.

STG_SHARE_DENY_WRITE = 20 (hex)

Does not allow any other to write to this element.

STG_SHARE_EXCLUSIVE = 10 (hex)

Does not allow any other to access this element in any way. **All child streams and storages must have this flag set.**

STG_SHARE_DENY_NONE = 40 (hex)

Allows any other to read from or write to this element as long as it is open.

either one of:

STG_DIRECT = 0

All changes are made directly to the file. This is faster and requires less memory, but is less flexible. **All streams must have this flag set.`**

STG_TRANSACTED = 10000 (hex)

All changes are made to an internally kept copy of the storage. It is possible to undo any changes made since opening the storage or the last **Commit()** by using the **Revert** method. The parent of a storage that is transacted need not be transacted itself.

if creating a new storage or stream (including a new Compound File) either one of:

STG_CREATE = 1000 (hex)

Create the new storage or stream, writing over any file that has the same name (if such a file exists).

STG_FAILIFTHERE = 0

Create the new storage or stream unless one with the same name currently exists.

if creating a new Compound File, one can use:

STG_CONVERT = 20000 (hex)

This will convert any normal file with the name and path specified to be converted into a Compound File with a single stream labeled "CONTENTS". All of the data that the file held previously will be in the "CONTENTS" stream. No changes are made to the original file until the root storage is committed, so it is possible to read any file as if it was a Compound File without changing the original in any way.

STG_DELETEONRELEASE = 4000000 (hex)

Automatically deletes the file as soon as the object that represents it is released (that is, set equal to "Nothing"). Useful for temporary files.

if opening a Compound file, one can use:

STG_PRIORITY = 20000 (hex)

Opens files with a minimum of overhead. Useful in cases where you are opening a storage just to copy it to another location. Must be used in conjunction with STG_READ and STG_DIRECT. Do not keep a STG_PRIORITY storage open any longer than necessary, as it prevents the Compound File from having any changes committed.

List of all Possible Flag Combinations:

CreateStorageFile and **CreateStorageMemory**

```
STG_DIRECT or STG_SHARE_EXCLUSIVE or  STG_READWRITE
STG_DIRECT or  STG_SHARE_EXCLUSIVE or STG_WRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_WRITE
STG_TRANSACTED or STG_DENY_READ or STG_READWRITE
STG_TRANSACTED or STG_DENY_READ or STG_WRITE
```

CreateStorage

```
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_WRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_WRITE
```

CreateStream

```
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_WRITE
```

OpenStorageFile and **OpenStorageMemory**

```
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_WRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READ
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_WRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READ
STG_TRANSACTED or STG_DENY_WRITE or STG_READWRITE
STG_TRANSACTED or STG_DENY_WRITE or STG_WRITE
STG_TRANSACTED or STG_DENY_WRITE or STG_READ
STG_TRANSACTED or STG_DENY_READ or STG_READWRITE
STG_TRANSACTED or STG_DENY_READ or STG_WRITE
STG_TRANSACTED or STG_DENY_READ or STG_READ
```

```
STG_TRANSACTED or STG_DENY_NONE or STG_READWRITE
STG_TRANSACTED or STG_DENY_NONE or STG_WRITE
STG_TRANSACTED or STG_DENY_NONE or STG_READ
```

OpenStorage

```
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READ
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READ
```

OpenStream

```
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_WRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READ
```

When you open a sub-storage or a stream, it must be at least as restricted as its parent. For example, if the parent is read-only the child must be as well. An exception exists if the parent of a stream is opened in a transacted, read-only mode - then the stream can be opened in a mode that allows writes. However, no writes are recorded to disk when this happens.

SeekPosition Flag

For use with the Stream object functions **Get**, **Put**, **VariantGet**, **VariantPut**, **VariantGetEx, VariantPutEx, GetString**, **PutString**, **GetRecord**, **SetRecord, GetBlock, PutBlock**.

STG_SEEK_DONTMOVE = -1

For reading or writing at the current location of the seek pointer.

Seek Flags

For use with Stream object procedure **Seek**.

STG_STREAM_SEEK_SET = 0

Position parameter is a count in bytes from the front of the stream.

STG_STREAM_SEEK_CUR = 1

Position parameter is a count in bytes forwards from the current seek position.

STG_STREAM_SEEK_END = 2

This flag means that the **Position** parameter is a count in bytes backwards from the end of the stream.

Move Flags

For use with Storage object procedure **MoveElement.**

STG_MOVEMOVE = 0

MoveElement will erase the original element.

STG_MOVECOPY = 1

MoveElement will **not** erase the original.

Commit Flags

For use with **Commit** and **Flush**. You can specify STG_DEFAULT or any combination of the others.

STG_DEFAULT = 0

Writes to those sections of this storage that have changed, but does not touch those that have not. This flag should be generally employed, as it faster than using STG_OVERWRITE.

STG_OVERWRITE = 1

Writes all information held in this element, whether or not it has been modified. This will occasionally result in smaller files than using STG_DEFAULT. See **CompressStorageFile**.

STG_ONLYIFCURRENT = 2

Prevents multiple users of a storage from overwriting the other's changes. STG_ONLY-IFCURRENT commits changes only if no one else has made changes since the last time this user opened or committed this storage. If other changes have been made, the error STG_E_NOTCURRENT is sent. You can overwrite the changes by attempting another Commit with STG_DEFAULT set.

STG_DANGEROUSLYCOMMITMERELYTODISKCACHE = 4

Only commits to the disk cache.

Directory File Type

For use with the Storage object method **Directory**. These flags indicate the type of object returned.

STG_TYPE_STREAM = 2

The name belongs to a stream.

STG_TYPE_STORAGE = 1

The name belongs to a storage.

STG_TYPE_NONE = 0

The end of the directory has been reached. There are no more names.

Possible Errors

STG_E_FILENOTFOUND = 30002

The specified file does not exist in the specified directory. This could also refer to a storage or stream that does not exist within a certain Compound File.

STG_E_PATHNOTFOUND = 30003

The specified file path or storage path does not exist.

STG_E_TOOMANYOPENFILES = 30004

There is a finite limit as to how many files can be kept open at any given time. In Windows 3.1, the limit is twenty file handles, while the 32 bit operating systems can keep far more open. Compound files can take up to three file handles each. Close unused open files before trying to open this file again.

STG_E_ACCESSDENIED = 30005

The file, storage, or stream cannot be accessed. More than likely there are permissions set to prevent writing to the file or it is locked by another application. This is usually caused by attempting to **Commit** a read-only storage or **Flush** a read-only stream.

STG_E_INVALIDHANDLE = 30006

The memory handle does not point to a valid place in memory, and is probably uninitialized.

STG_E_INSUFFICIENTMEMORY = 30008

Compound files require various amounts of memory for buffering. If there is insufficient memory, certain functions (such as opening a storage) will not work. If you need to open a Compound File for saving and get this error, you can try to reduce your memory requirements by using the STG_DIRECT instead of STG_TRANSACTED. **Commit** and **Flush** are designed to work without taking up any

memory, so you can prevent this error by keeping a Compound File open throughout the time it is used by your program.

STG_E_INVALIDPOINTER = 30009

An incorrect parameter was passed to a method.

STG_E_NOMOREFILES = 30018

There is a finite limit as to how many files can be kept open at any given time. In Windows 3.1, the limit is twenty file handles, while the 32 bit operating systems can keep far more open. Compound files can take up to three file handles each. Close unused open files before attempting to open this file again.

STG_E_DISKISWRITEPROTECTED = 30019

The disk cannot be written to. Disable disk write protection before trying again.

STG_E_SEEKERROR = 30025

An attempt was made to seek past the end of the stream.

STG_E_WRITEFAULT = 30029

A problem was encountered while trying to write information to the Compound File. The file may be corrupted, or there may be a problem with the storage medium.

STG_E_READFAULT = 30030

A problem was encountered while trying to read information from the Compound File. The file may be corrupted, or there may be a problem with the storage medium.

STG_E_SHAREVIOLATION = 30032

An attempt was made to access a section of the Compound File that is being used by another program. Try again later. This error sometimes happens after a program that was reading a Compound File crashes or exits without properly closing the file.

STG_E_LOCKVIOLATION = 30033

An attempt was made to access a section of the Compound File that has been locked by another program. Try again later. This error may occur after a program that was reading a Compound File crashes or exits without properly closing the file.

STG_E_FILEALREADYEXISTS = 30080

This operation would have overwritten an already existing file, and was therefore terminated. See the STG_CREATE flag to find out how to purposefully overwrite an existing file.

STG_E_INVALIDPARAMETER = 30087

A method parameter was of the incorrect type or held invalid information.

STG_E_MEDIUMFULL = 30112

There is no more space on this disk or partition. Try another disk or partition, or try using the STG_OVERWRITE flag.

STG_E_ABNORMALAPIEXIT = 30250

An internal OLE function failed.

STG_E_INVALIDHEADER = 30251

The Compound File is probably corrupt, or was written by a program that did not fully adhere to the Structured Storage specifications.

STG_E_INVALIDNAME = 30252

The name you chose for this storage or stream is invalid. Names are limited to thirty two characters, and cannot include the characters "\", "/", ":", "!", "." or "..".

STG_E_UNKNOWN = 30253

An internal undetermined error occurred.

STG_E_INVALIDFLAG = 30255

An illegal combination of flags or an incorrect flag was used. Review the documentation and confirm flags may be used in conjunction with others.

STG_E_NOTCURRENT = 30257

Another opening of the Storage object has modified it. If you want to **Commit** anyway, overwriting any changes, try using the STG_OVERWRITE flag.

STG_E_REVERTED = 30258

You have attempted to access a section of the Compound File that no longer exists because the Compound File has been reverted to its original status.

STG_E_CANTSAVE = 30259

Cannot save for reasons other than lack of access or lack of space.

STG_E_OLDFORMAT = 30260

The file you have attempted to read is an OLE 1.0 DocFile, and is incompatible with the Structured Storage format.

STG_E_OLDDLL = 30261

Your OLE DLL's' are probably out of date. It is likely that some software that was recently installed overwrote the correct ones.

Possible Component Errors

Because of the nature of ATL components, the dwStg.dll component will usually trigger standard Visual Basic and .NET errors when something goes wrong. For example, when you are attempting to open a file that is already opened, you will get the error "Path/Access error" (error 75 in Visual Basic). This is the same error you would get in similar regular file access functions. See the Visual Basic or .NET documentation for other errors related to file access.

There are still some errors that are unique to the dwStg.dll component. They are listed below.

STG_E_INVALIDHEADER = 800300FB (hex)

The file is not a standard structured storage file. It may have been written using a non-standard implementation of the structured storage interfaces, or it may be corrupt.

STG_E_INVALIDNAME = 800300FC (hex)

The name of the stream or storage given is not valid. Names are limited to thirty two characters, and cannot include the characters “\”, “/”, “:”, “!”, “.” or “..”.

STG_E_NOTCURRENT = 80030101 (hex)

The storage has been changed since the last commit. You will only get this error if you try to **Commit** using the STG_ONLYIFCURRENT flag, and there have in fact been changes.

STG_E_REVERTED = 80030102 (hex)

This error means that the action you are trying to do cannot be completed because the dStorage or dStream object has been invalidated by a **Revert** in a storage above your object.

STG_E_CANTSAVE = 80030103 (hex)

The object you are trying to persist using dStorage.**PutObject** or dStream.**PutObject** has reported that it cannot save itself.

STG_E_DOCFILECORRUPT = 80030109 (hex)

The file has been corrupted and cannot be read. There is a bug in ole32.dll that can cause this error in large files (over 80mb) – be sure to use version 4.0 or later.

STG_E_NULLPOINTER = 32112

This error is caused by sending a null pointer to PutPicture.

STG_E_RECORDEOF = 32113

This error is sent when you try to access a record past the end of the stream using the GetRecord method.

Storage Browser (StgBrwsr.exe)

Storage Browser is a program that allows you to peek into Compound Files and see how they are organized. Several Microsoft programs (for example, Word for Windows 6.0) already use Structured Storage for their files. Storage Browser will run under Windows 95, Windows 98, Windows NT and Windows 3.1. The Visual Basic source code for this program is included, and may be used in your own projects.

Figure 1 depicts the Main Window of Storage Browser.

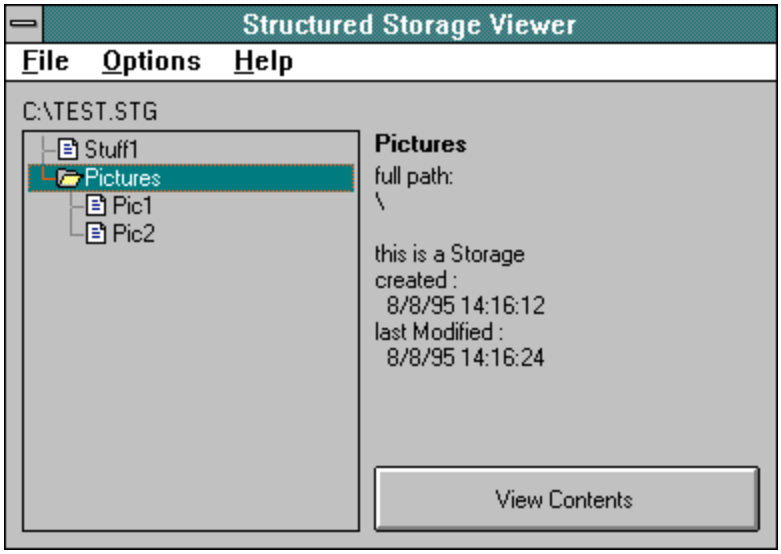

Figure 1
The Primary Window of the Storage Browser

The box on the left depicts a hierarchical list of the elements in the file. Storages are identified by file folder icons, and streams by document icons. By double-clicking on a storage, the elements contained in the storage are shown or hidden. Above this box is the name and path of the file being browsed.

When an element is highlighted, information about that element is presented in the area on the right. This information includes the name, type, size (if it is a stream), date (if it is a storage), and purpose (if the name begins with a special character).

To view the contents of a stream, select it and click the **View Contents** button on the lower right. If the element selected is the Summary Information stream or Document Summary Information stream, a window containing information about the document will appear. Otherwise, it will bring up the window titled **View of Stream**:

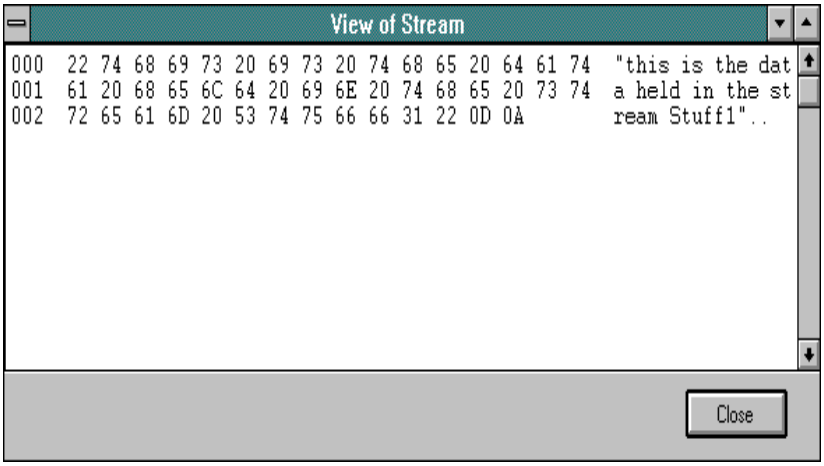

Figure 2
View of Stream Window

The data is always shown in binary format. The column on the extreme left is the number of the row. The column in the middle depicts the contents in hexadecimal form. The column on the extreme right presents the same information as the second column, but as characters.

Storage Menu

The first menu entry is **Load Compound File**. This will bring up the File Open common dialog box, which will allow you to choose the file you would like to browse. If you choose a file that is not a Compound File, the program will give you the option of viewing it anyway. The **Save Changes** menu entry lets you permanently save any changes you made to the Compound File.

The menu entry **Compress Compound File** also brings up the File Open command dialog box. Select the Compound File that you wish to compress. (See the description of the **CompressStorageFile** method for an explanation as to when and why you would wish to use this function.)

Options Menu

The following commands operate on the currently selected element.

Command	Description
Rename Element	Brings up a dialog box that allows you to type in a new name for the element.
Delete Element	Erases the currently selected element, including any streams and storages it might contain.

Help Menu

This menu allows you to access the about box.

The Desaware File Property Component

Since the introduction of the structured storage format, many programs use it as the basis of the files they save to disk. Most applications, however, still use their own proprietary format. This is a problem because one of the key features of structured storage files – the ability to have a commonly readable table of file properties – is most useful if all documents have them. Otherwise, searching for files with certain authors or word counts will miss all those files without these properties. In response to this, Microsoft has introduced the ability to add many of these file properties to any file located on a Windows 2000 NTFS formatted drive. The Desaware Property control is designed to let you read and write file properties for files on such a drive, as well as structured storage files like Office documents on any drive or operating system.

Technology of File Properties

Structured storage files consist of blocks of data (called streams) held in hierarchical directories (called storages). At first, a single stream called "SummaryInformation" held file properties. Because this information was only being used by Microsoft Word at the time, it only held properties that related to word processor documents. As other Microsoft Office applications started using the structured storage format, these were no longer sufficient. Microsoft added a second stream, called "DocumentSummaryInformation", which added new properties specific to other Office products and a section for user-defined properties. The Desaware File Property Component, like the Desaware Storage Component, can read and write any of these properties, and can add them to any structured storage file that does not have them.

Every implementation of the NTFS disk format has supported multiple "streams" (not to be confused with structured storage streams) of information per file, although this is a little known or used feature. A NTFS stream is another section of the file, but it is not shown in the size of the file in Explorer, nor is it seen if you look at a file in a text or binary editor. You can experiment with streams on a command line. The following line will copy a list of files in the current directory into a new text file entitled "directory.txt".

```
dir > directory.txt
```

The next line will do the same, but the list of files will go into the stream named "section2"

```
dir > directory.txt:section2
```

Note how the file has not grown in size (at least according to the `dir` command or Explorer), and how Notepad only shows the single list. You can see what is in the "`section2`" stream by using the following command line:

```
more < directory.txt:section2
```

Windows 2000 uses NTFS streams to save file properties for any file, so setting properties for a file does not change its internal data. However, note that not all Microsoft operating systems support streams. If you copy a file to a non-NTFS drive you will lose the file properties. If you use an operating system that does not know about streams, like Windows 95, to copy a file on an NTFS drive, you will also lose any streams, even if the destination drive is also NTFS formatted. Windows 2000 only stores those properties found in the SummaryInformation stream, so you cannot add custom properties to non-structured storage files on NTFS drives. For more information on NTFS file streams, see article Q105763 in the Microsoft Developer's Network CDROM or on the Microsoft web site.

File Property Component Example

The following is an example of how you would use the File Property Component to read and write file properties:

```
' VB
' Make sure that "Desaware Property component"
' is selected in the Visual Basic references dialog.
' Create an instance of the Property component.
Dim dwProp As New dwProperty

' This determines if the file is a structured
' storage file or not.
IsStorage = dwProp.IsStorageFile ("C:\file.txt")

' Open a file.  This should be a structured storage
' file, or a file on a Windows 2000 NTFS drive.  If
' it is not one of these, you will get a runtime
' error.
dwProp.OpenFile "C:\file.txt", STG_FLAGS_NORMAL
```

```
' Read the summary information from the file.
If (dwProp.ReadSummaryInfo = True) Then
    ' get the title of the document
    titleString = dwProp.GetTitle()
    ' add something to the string
    titleString = titleString & " version 2"
    ' set the new title
    dwProp.SetTitle titleString
    ' save the changes to the
    ' SummaryInformation stream
    dwProp.SaveSummaryInfo
    ' when done making changes, release the file.
    dwProp.ReleaseFile
End If

' VB.NET
' Make sure that "Desaware Property component"
' is selected in the Visual Basic references dialog.
' Create an instance of the Property component.
Dim Prop As dwProperty = New dwProperty()

' This determines if the file is a structured
' storage file or not.
IsStorage = Prop.IsStorageFile ("C:\file.txt")

' Open a file.  This should be a structured storage
' file, or a file on a Windows 2000 NTFS drive.  If
' it is not one of these, you will get a runtime
' error.
Prop.OpenFile ("C:\file.txt",
   dwPropConsts.STG_FLAGS_NORMAL)

' Read the summary information from the file.
If (Convert.ToBoolean (Prop.ReadSummaryInfo ()) =
   True) Then
    ' get the title of the document
    titleString = Prop.GetTitle()
    ' add something to the string
    titleString = titleString & " version 2"
    ' set the new title
    Prop.SetTitle (titleString)
    ' save the changes to the
    ' SummaryInformation stream
    Prop.SaveSummaryInfo()
    ' when done making changes, release the file.
    Prop.ReleaseFile()
End If
```

```
// C#
// Make sure that "Desaware Property component"
// is selected in the Visual Basic references
// dialog. Create an instance of the Property
// component.
dwProperty Prop = new dwProperty();

// This determines if the file is a structured
// storage file or not.
IsStorage = Prop.IsStorageFile ("C:\file.txt");

// Open a file.  This should be a structured storage
// file, or a file on a Windows 2000 NTFS drive.  If
// it is not one of these, you will get a runtime
// error.
Prop.OpenFile ("C:\file.txt",
   dwPropConsts.STG_FLAGS_NORMAL);

// Read the summary information from the file.
if (Convert.ToBoolean (Prop.ReadSummaryInfo()) ==
   true)
{
    // get the title of the document
    titleString = dwProp.GetTitle();
    // add something to the string
    titleString = titleString + " version 2";
    // set the new title
    Prop.SetTitle (titleString);
    // save the changes to the
    // SummaryInformation stream
    Prop.SaveSummaryInfo();
    // when done making changes, release the file.
    Prop.ReleaseFile();
}
```

File Property Component Properties

*ComponentObject**.Filename***

This property holds the name of the file that is currently open. If no file is currently open, it is blank.

File Property Component Methods

Note for .Net Users:

All function descriptions below use the Visual Basic standard of "Long" for a 32 bit integer. In .NET languages, the correct type would be "Integer" in VB.NET or "int" in C#. They also use the Visual Basic data type "Date". The .NET equivalent of that type is "DateTime".

Note that .NET does not correctly recognize the Boolean type of the return value in OLE components. You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function. Convert is an object in the "System" name space – see the .NET documentation for more information.

*ComponentObject.**OpenFile***
(Filename as String, Flags as Long)

You must first open a file before you can read or write properties to it. The **Filename** parameter is the path and name of the file to be opened. The **Flags** parameter describes the way the file is opened. Usually the constant STG_NORMAL is used here – see the section on constants for more information. When **OpenFile** is called on a normal file on an Windows 2000 NTFS drive, it does not keep the file open. If the file is opened or written to by another application after **OpenFile** is called and before other Property control methods are called no problem will occur. If the file is moved or deleted after **OpenFile** is called, you will get an error on calling **ReadSummaryInformation** or **SaveSummaryInformation**.

*ComponentObject.**ReleaseFile ()***

When you are done accessing a file, call the **ReleaseFile** method. This will clear certain internal buffers, and make sure the file is written to disk.

*ComponentObject.**ReadSummaryInfo () as Long***

Retrieves information from the SummaryInformation property set stream in the root storage of this Compound File. If there is no SummaryInformation property set, **ReadSummaryInfo()** generates an error. Before the **ReadSummaryInfo()** is called, using any function beginning with “Get” will return the default value, which will be either

zero or a blank string depending on the type. The return value is a bitmap of the streams which the file contains.

Comparing the return value to the constants SUM_INFO, DOC_INFO and USR_INFO will tell you if the file contains a SummaryInformation stream, a DocumentSummaryInformation stream, and User-defined Summary Information respectively. Here is a sample of how to use this feature:

```
' VB and VB.NET
' Using the And operator will return a non-zero
' value if there are bits in the return value and
' bits in DOC_INFO that match.  See the Visual
' Basic documentation for "And" for more information
retval = dwProp.ReadSummaryInfo()
If (retval And DOC_INFO) then
    ' yes, this file has Document summary info
End If

// C#
retval = dwProp.ReadSummaryInfo();
if (retval & dwPropConsts.DOC_INFO)
{
    // yes, this file has Document summary info
}
```

*ComponentObject.**SaveSummaryInfo ()***

Saves any changes to the SummaryInformation. If you have opened a structured storage file, these changes may not be written to disk until the **ReleaseFile** method is called.

*ComponentObject.**IsStorageFile (Filename as String) as Boolean***

When passed a path and name of a file in the **Filename** parameter, this will return True if the file is a structured storage file, and False if it is not. Only compare the return value against False, as different programming languages have different values for True.

*ComponentObject.**EnableComponent (LicenseKey as String)***

This method allows you to use the File Property Component in your own Visual Basic components, such as ActiveX controls or ActiveX documents. Otherwise, the File Property Component will also think it is being used in Visual Basic design time, and will report a license error.

You first create a key by using the dwLicGen.exe program. This program will ask you to enter the compiled name of your component (not including a path, but including an extension). Each key is uniquely created from the component name. If you change your component name, you will have to create a new license key. After the program generates the key, you will have the option to copy it to the clipboard to post into your application.

You then call the **EnableComponent** method before you access any of the components methods, like in the Initialize or Sub Main functions. If you ever unload or remove the last reference to the File Property Component, you will have to go through this process again upon creating a new reference.

```
' VB
Private Sub UserControl_Initialize()
' Your key will be different.
   DwStg.EnableComponent "2E3499248E322D3636853B53"
End Sub

' VB.NET //C#
' Your key will be different.
DwStg.EnableComponent ("2E3499248E322D3636853B53")
```

*ComponentObject.**IsValid () as Boolean***

Returns True if a file has been opened, False if not. Only compare the return value against False, as different programming languages have different values for True. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

ComponentObject.***ConvertToLocalTime (InputTime as Date) as Date***

This will convert a date from Greenwich Mean Time (**GMT**) to local time. Times in SummaryInformation properties are stored in **GMT** (this is done automatically by the properties of the component that use dates).

Summary Information Methods

The following methods all deal with the properties in the SummaryInformation stream. If the file does not have such a stream when it is opened, one will be created on calling SaveSummaryInformation.

*ComponentObject.**SetTitle (Title as String)***

Sets the title of the document.

*ComponentObject.**GetTitle () as String***

Returns the title of the document.

*ComponentObject.**SetSubject (Subject as String)***

Sets the subject of the document.

*ComponentObject.**GetSubject () as String***

Returns the subject of the document.

*ComponentObject.**SetAuthor (Author as String)***

Sets the author of the document.

*ComponentObject.**GetAuthor () as String***

Returns the author of the document.

*ComponentObject.**SetKeywords (Keyword as String)***

Sets key words related to the subject of the document. These key words are typically used in search routines.

*ComponentObject.**GetKeywords () as String***

Returns key words relating to the subject of the document.

*ComponentObject.**SetComments (Comment as String)***

Sets the comment field. This area might be used for making notes to oneself or others.

*ComponentObject.**GetComments () as String***

Returns the comment field.

*ComponentObject.**SetLastAuthor (LastAuthor as String)***

Sets the latest person to have modified the document.

*ComponentObject.**GetLastAuthor () as String***

Returns the last person who has modified the document.

*ComponentObject.**IncrementRevNum ()***

Increments the number of times a document has been revised by one.

*ComponentObject.**SetRevNum (TimesRevised as Long)***

Sets the number of times the document has been revised.

*ComponentObject.**GetRevNum () as Long***

Returns the number of times the document has been revised.

*ComponentObject.**StartEditTimer ()***

This starts a timer within the dwProp component. It is useful for keeping track how long a document has been opened, so you can update the TotalEditTime property. You can call this method when the file is first opened, and the **AddEditTimerToTotal** method when the file is saved.

*ComponentObject.**AddEditTimerToTotal ()***

Adds the amount of time since **StartEditTimer** was called to the property recording the total amount of time the document has been opened.

*ComponentObject.**GetTotalEditTime () as Long***

Returns the total amount of time a document has been open, in minutes.

*ComponentObject.**SetTotalEditTime (Minutes as Long)***

Sets the total amount of time a document has been open, in minutes.

*ComponentObject.**RecordPrintDate ()***

Sets the field containing the last time this document was printed to the current date and time.

*ComponentObject.**GetLastPrintDate () as Date***

Returns the last time the document was printed.

*ComponentObject.**RecordCreateDate ()***

Sets the field containing the time the document was created to the current date and time.

*ComponentObject.**GetCreateDate () as Date***

Returns the time the document was created.

*ComponentObject.**RecordSaveDate ()***

Sets the field containing the last time this document was saved to the current date and time.

*ComponentObject.**GetLastSaveDate () as Date***

Returns the last time the document was saved.

*ComponentObject.**SetNumberOfPages (NumPages as Long)***

Sets the number of pages.

*ComponentObject.**GetNumberOfPages () as Long***

Returns the number of pages.

ComponentObject.***SetNumberOfWords (NumWords as Long)***

Sets the number of words.

ComponentObject.***GetNumberOfWords () as Long***

Returns the number of words.

ComponentObject.***SetNumberOfCharacters (NumChars as Long)***

Sets the number of characters.

ComponentObject.***GetNumberOfCharacters () as Long***

Returns the number of characters.

ComponentObject.***SetApplication (AppName as String)***

Sets the name of the program that created the document.

ComponentObject.***GetApplication () as Long***

Returns the title of the program that created the document.

ComponentObject.***SetTemplate (Template as String)***

Sets the filename of the template used in the document.

ComponentObject.***GetTemplate () as String***

Returns the filename of the template used in the document.

ComponentObject.***SetSecurity (SecurityLevel as Long)***

Sets the recommended security level of the document. Note that this does not actually supply any security, but only serves as a reminder to the application that reads the file.

Value	Security Procedure
0	None.
1	Password protected. Do not allow viewing or editing of this document. Other programs cannot view or edit this document.
2	Read-Only recommend. The user will be warned if there is any attempt to edit the document.
3	Read-Only enforced. Does not allow any changes to the document.
4	Locked for Annotations. Does not allow any changes to the document.

*ComponentObject.**GetSecurity () as Long***

Returns the security level of the document. Your program should behave as follows:

Value	Security Procedure
0	None.
1	Password protected. Do not allow viewing or editing of this document unless you know what type of password protection is involved, and the password given is correct.
2	Read-Only recommend. Warn the user if there is any attempt to edit the document.
3	Read-Only enforced. Do not allow any changes to the document.
4	Locked for Annotations. Do not allow any changes to the document.

Document Summary Information Methods

These methods access the DocumentSummaryInformation stream. Not every structured storage file that contains a SummaryInformation stream contains a DocumentSummaryInformation stream. If you open a structured storage file that only has a Summary Information stream and you use any of the Document Summary Information methods that begin with "Set", a DocumentSummaryInformation stream will be added to the file. Normal files on a Windows 2000 NTFS drive do not have these properties – only the those which have Summary Information. If you open a normal file on a Windows 2000 NTFS drive and use a Document Summary Information method that begins with "Set", it will have no effect.

*ComponentObject.**SetScaleCrop (Scale as Boolean)***

Set this to True if this document is supposed to be scaled to the dimensions of the window, False if it is to be cropped to the dimensions of the window.

*ComponentObject.**GetScaleCrop () as Boolean***

Returns True if this document is supposed to be scaled to the current dimensions of the window, False if it is to be cropped to the dimensions of the window. Only compare this to "False" in Visual Basic, as True can refer to any non-zero number. IBYou will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

*ComponentObject.**SetLinksUpToDate (Links as Boolean)***

Set this to True if all the links are up to date. Links may refer to internal links in the document or links to web sites depending upon the application.

*ComponentObject.**GetLinksUpToDate () as Boolean***

Returns True if all the links are up to date. Only compare this to "False" in Visual Basic, as True can refer to any non-zero number. ***Note that .NET does not correctly recognize Boolean return types.*** You will need to convert the type back to a Boolean type using the Convert.ToBoolean () function.

*ComponentObject.**SetCategory (Category as String)***

Sets the category of the document.

*ComponentObject.**GetCategory () as String***

Returns the category of the document.

*ComponentObject.**SetPresentationTarget (Target as String)***

Sets the presentation target of the document. Used in Microsoft PowerPoint.

*ComponentObject.**GetPresentationTarget () as String***

Returns the presentation target of the document.

*ComponentObject.**SetManager (Manager as String)***

Sets the manager of the document writer.

*ComponentObject.**GetManager () as String***

Returns the manager of the document writer.

*ComponentObject.**SetCompany (Company as String)***

Sets the company with which the document is associated.

*ComponentObject.**GetCompany () as String***

Returns the company with which the document is associated.

*ComponentObject.**SetNumBytes (NumBytes as Long)***

Sets the size of the document in number of bytes. Usually refers to the section of the document edited by the user, not the entire size of the file.

*ComponentObject.**GetNumBytes () as Long***

Returns the size of the document in number of bytes.

*ComponentObject.**SetNumLines (NumLines as Long)***

Sets the number of lines in the document.

*ComponentObject.**GetNumLines () as Long***

Returns the number of lines in the document.

*ComponentObject.**SetNumParagraphs (NumParas as Long)***

Sets the number of paragraphs in the document.

*ComponentObject.**GetNumParagraphs () as Long***

Returns the number of paragraphs in the document.

*ComponentObject.**SetNumSlides (NumSlides as Long)***

Sets the number of slides in the document.

*ComponentObject.**GetNumSlides () as Long***

Returns the number of slides in the document.

*ComponentObject.**SetNumNotes (NumNotes as Long)***

Sets the number of notes in the document.

*ComponentObject.**GetNumNotes () as Long***

Returns the number of notes in the document.

*ComponentObject.**SetNumHiddenSlides (NumHiddenSlides as Long)***

Sets the number of hidden slides in the document.

*ComponentObject.**GetNumHiddenSlides () as Long***

Returns the number of hidden slides in the document.

*ComponentObject.**SetNumMMClips (NumClips as Long)***

Sets the number of multimedia clips in the document.

*ComponentObject.**GetNumMMClips () as Long***

Returns the number of multimedia clips in the document.

User-Defined Property Methods

These methods access the user-defined portion of the DocumentSummaryInformation stream. Not every structured storage file that contains a SummaryInformation stream contains a DocumentSummaryInformation stream. Normal files on a Windows 2000 NTFS drive do not have these properties.

Here is an example of how you would use these methods to read and print a list of existing properties:

```
' VB 5 and 6
Dim count as Long
Dim v as Variant
Dim i as Long

' if the user-defined part of the
' DocumentSummaryInformation stream
' exists, load its information
count = 0
If (dwProp.OpenSummaryInfo And USR_INFO) Then
    ' get the number of properties
    count = dwProp.UserCount
    ' print all the property names and data
    For i = 0 To count - 1
        Debug.Print "name:", dwProp.UserDirectory(i)
        Debug.Print "data:", dwProp.UserGet(i)
    Next i
```

The following is an example of how you would add a number and a string into the Document Summary Information property set:

```
    ' Add a property that contains a number
    v = 1234
    count = dwProp.UserAdd("NewEntry1", v)
    ' Add a property that contains a string
    v = "String Value"
    count = dwProp.UserAdd("NewEntry2", v)

    ' This saves all summary information.
    dwProp.SaveSummaryInfo
    ' make sure the summary information is
    ' actually written to disk
    dwProp.ReleaseFile
End If
```

```
' VB.NET
Dim count as Integer
Dim v as Object
Dim i as Integer

' if the DocumentSummaryInformation stream
' exists, load its information
count = 0
If (Prop.OpenSummaryInfo() And
   dwPropConsts.USR_INFO) Then
    ' get the number of properties
    count = Prop.UserCount()
    ' print all the property names and data
    For i = 0 To count - 1
        MessageBox.Show ("name:" &
   Prop.UserDirectory(i) & " data:" &
   Prop.UserGet(i))
    Next i

    ' Add a property that contains a number
    v = 1234
    count = Prop.UserAdd("NewEntry1", v)
    ' Add a property that contains a string
    v = "String Value"
    count = Prop.UserAdd("NewEntry2", v)

    Prop.SaveSummaryInfo()
    ' just like any stream, you must commit the
    ' root storage to actually write to disk
    Prop.ReleaseFile()
End If

// C#
int countv;
object v;
int i;

// if the DocumentSummaryInformation stream
// exists, load its information
count = 0;
if (Prop.OpenSummaryInfo() & dwPropConsts.USR_INFO)
{
    // get the number of properties
    count = Prop.UserCount();
    // print all the property names and data
    for (i = 0; i > count; i++)
    {
```

```
        MessageBox.Show ("name:" &
    Prop.UserDirectory(i) & " data:" &
    Prop.UserGet(i));
     }

     // Add a property that contains a number
     v = 1234;
     count = Prop.UserAdd("NewEntry1", v);
     // Add a property that contains a string
     v = "String Value";
     count = Prop.UserAdd("NewEntry2", v);

     Prop.SaveSummaryInfo();
     // just like any stream, you must commit the
     // root storage to actually write to disk
     Stg.ReleaseFile();
}
```

*ComponentObject.**UserSet***
(PropertyID as Long, Name as String, Data as Variant)

Modifies the existing property specified by **PropertyID**. **Data** can be either an integer, a date, a string, or a boolean value. Regular integers are converted to long integers to match the existing standard. If **PropertyID** does not correspond to an existing property, an error is generated. Due to certain limitations in Windows, dates must be between the years 1980 and 2107. If you need to store a date outside this range, you can use a string variant.

*ComponentObject.**UserGet***
(PropertyID as Long) as Variant

Returns the data associated with the property specified by **PropertyID**.

*ComponentObject.**UserAdd***
(Name as String, Data as Variant) as Long

Add a new property to the end of the list of user-defined properties in this Document Summary Information stream. The new count of user-defined properties is returned. Due to certain limitations in Windows, dates must be between the years 1980 and 2107 – if you have to store a date that is outside this range, you can use a string.

```
' VB
' Add a string
Dim v as Variant
v = "New String Data"
count = dwProp.UserAdd ("value name", v)

' VB.NET
Dim v as Object
count = dwProp.UserAdd ("value name", v)

// C#
object v;
count = dwProp.UserAdd ("value name", v);
```

ComponentObject.***UserCount () as Long***

Returns the number of user-defined properties in this Document Summary Information stream.

ComponentObject.***UserDelete (PropertyID as Long) as Long***

This deletes the specified property. After the deletion, all properties are renumbered to remove the empty space. The new count of user-defined properties is returned.

```
' Delete the last property
count = dwProp.UserCount ()
count = dwProp.UserDelete (count - 1)
```

ComponentObject.***UserDirectory (PropertyID as Long) as String***

Returns the name of the specified property.

File Property Component Constants

OpenFile flags parameter

any one of:

STG_READ = 0

Read from, but not write to the file. Methods beginning with "Set" will still work, but an attempt to call **SaveSummaryInformation** will result in an error. With STG_DIRECT set, this is faster and takes less memory and disk space using other flag combinations.

STG_WRITE = 1

Write to, but not read from the element.

STG_READWRITE =2

Allows both reading from and writing to the element.

any one of:

STG_SHARE_DENY_READ = 30 (hex)

Does not allow any other to read from this element.

STG_SHARE_DENY_WRITE = 20 (hex)

Does not allow any other to write to this element.

STG_SHARE_EXCLUSIVE = 10 (hex)

Does not allow any other attempt to access this file in any way from the time it is opened to when the **ReleaseFile** method is called. If the file is a normal file on a Windows 2000 NTFS drive, it must have this flag selected; however, such files are only accessed at the time of reading or writing.

STG_SHARE_DENY_NONE = 40 (hex)

Allows any other to read from or write to this file as long as it is open.

either one of:

STG_DIRECT = 0

All changes are made in a more direct fashion (although changes are not always written directly to the file, depending on the version of OLE installed). This is faster and requires less memory. If the file is a normal file on a Windows 2000 NTFS drive, it must have this flag selected

STG_TRANSACTED = 10000 (hex)

All changes are made to an internally kept copy of the storage. This is more memory intensive, but allows a greater array of flag choices. It also provides other advantages that do not relate to summary information sets.

Or you can just use:

STG_NORMAL

STG_NORMAL is a combination of STG_SHARE_ EXCLUSIVE, STG_DIRECT, and STG_READ-WRITE. This is the most common flag combination, and will let you access any file that is not already open.

List of all possible flag combinations:

```
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_WRITE
STG_DIRECT or STG_SHARE_EXCLUSIVE or STG_READ
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READWRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_WRITE
STG_TRANSACTED or STG_SHARE_EXCLUSIVE or STG_READ
STG_TRANSACTED or STG_DENY_WRITE or STG_READWRITE
STG_TRANSACTED or STG_DENY_WRITE or STG_WRITE
STG_TRANSACTED or STG_DENY_WRITE or STG_READ
STG_TRANSACTED or STG_DENY_READ or STG_READWRITE
STG_TRANSACTED or STG_DENY_READ or STG_WRITE
STG_TRANSACTED or STG_DENY_READ or STG_READ
STG_TRANSACTED or STG_DENY_NONE or STG_READWRITE
STG_TRANSACTED or STG_DENY_NONE or STG_WRITE
STG_TRANSACTED or STG_DENY_NONE or STG_READ
```

The Registry

One of the key features of Microsoft Windows is its ability to be customized to suit the user. Not only can a user specify color schemes or pictures on the desktop, but also programs can remember where they were last placed and which documents were most recently used. Microsoft originally specified that Windows applications keep configuration information inside a text file called **WIN.INI**. This created a standard for programmers, while allowing applications and users to change program and system settings directly. As the number of Windows applications exploded, limitations in this standard became apparent. Initialization files became large and—because of their flat arrangement—hard to understand and edit. With the release of Windows 3.0, Microsoft tried to alleviate the first problem by creating API functions for handling private .INI files. This, of course, created problems of its own, —filling the users \WINDOWS directory with countless .INI files and making it difficult for applications to communicate with one another.

The beginning of the solution was introduced in Windows 3.1. With the introduction of inter-process forms of communication such as Dynamic Data Exchange (DDE) and Object Linking & Embedding (OLE), it became vital to create a centralized database of program descriptions. The **Registration Database** is the result. It has a hierarchical structure, allowing complex data structures to be stored cleanly. It is in binary format for quicker access. Besides holding descriptions of DDE and OLE entry points, it also contains associations between extensions and applications, user readable names for programs and files, and other useful information.

The main elements of the hierarchical structure are **keys**. Keys can contain other keys. The key at the root of this structure is named **HKEY_CLASSES_ROOT**. Any key can have one **value** associated with it. This value is in the form of a string of characters.

Figure 3 illustrates how a portion of the Registration Database might be structured.

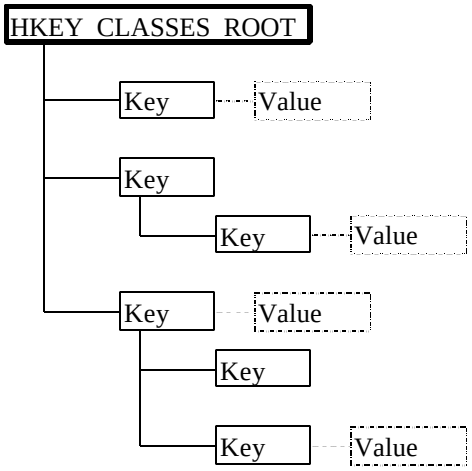

Figure 3
Registration Database Structure

Because the registry is in binary format, it cannot simply be loaded into a text editor as is customary with initialization files. Microsoft provides a program called REGEDIT, which allows the user to view and modify the Registration Database. Running REGEDIT with the command line flag "-v" turns on the advanced editing mode that reveals more information from the registration database.

However, the Registration Database was (and still is) hampered by the 64K size limit. Additionally, the Registration Database is limited to one file (REG.DAT), so it cannot be split when the size limit is reached. It also has a limited purpose, and could not replace the .INI file as an application configuration storage.

One of Microsoft's priorities in designing Windows NT was to overcome limitations discovered in 16 bit Windows. To solve the problems with initialization, Microsoft programmers created the **Registry**. At its heart, it is similar to the Registration Database -- but much more useful. Instead of the lone HKEY_CLASSES_ROOT root key, the Registry has four root keys -- of which HKEY_CLASSES_ROOT is one. These new keys provide the structure and the space for any configuration information a program might need to store. There is also space to locate version information and other data a program might want to make public. The whole database can be of almost unlimited size. It is specially formatted and buffered to allow extremely fast access. Keys can now hold any number of named values, and these values can be any of a number of different data types. The .INI files became obsolete -- in fact, when 16 bit applications make API calls to read or write information to system .INI files, Windows NT actually re-routs them to a special location inside the Registry. The 32 bit version of the Registry editor is named REGEDT32.EXE. (You might not find it on your machine. System administrators do not always make this program available to end users because of the possibility of accidental destruction of important Windows data.)

Figure 4 depicts a representation of a portion of the Registry.

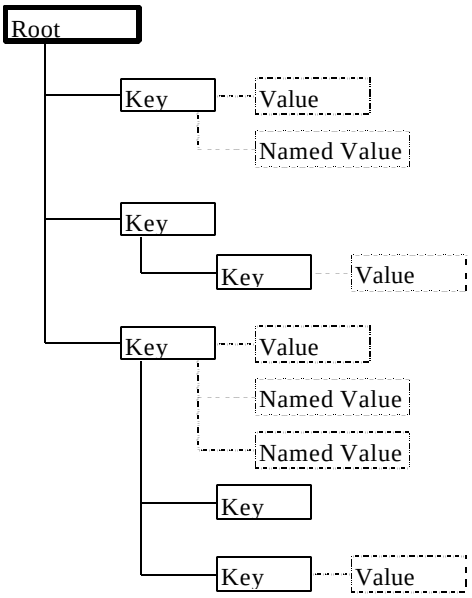

Figure 4
The Registry

Windows 95/Windows 98 also depends heavily upon the Registry, although not quite as much as Windows NT. Windows 95/Windows 98 does not need WIN.INI and SYSTEM.INI, but will use them if they exist. Any attempt to change these initialization files will not result in a change in the Registry. Windows 95 added two new root keys to the NT Registry while removing areas devoted to security and other Windows NT-only features. The Windows 95/Windows 98 Registry editor is named REGEDIT.EXE.

Keys and Values

The objects that make up the hierarchical structure of the Registry are called **keys**. Key names cannot include spaces, backslashes ("\"), asterisks ("*") or question marks. Except for those keys representing file extensions, key names cannot begin with periods. The name of a key must be unique with respect to its parent key. Keys can contain information in the form of **values**. Every key has one unnamed value, called the default value. This value is generally in the form of a string. In the Windows 95, Windows 98 and Windows NT operating systems, any number of named values can also exist in a key. The information in a named value can in one of many forms. A value can hold up to one megabyte of data. See the *Value Data Types* section later in this manual.

Root Keys

HKEY_CLASSES_ROOT

Available in Windows 3.1, Windows NT and Windows 95 and Windows 98. It is mapped to HKEY_LOCAL_MACHINE\Software\Classes in all but Windows 3.1, where it is the only root available. It is where file extensions are mapped to applications, DDE links are described and OLE objects are defined. Keys in this root generally do not have any named values, and all values are plain strings in order to be compatible with the Windows 3.1 Registration Database.

HKEY_CURRENT_USER

Available in Windows NT, Windows 95 and Windows 98. It contains the currently-logged-in user's settings. It can be used to save settings that are particular to individuals. It is mapped to the key in HKEY_USERS that belongs to the current user.

HKEY_LOCAL_MACHINE

Available in Windows NT, Windows 95 and Windows 98. It has information about the system and non-user specific information about software. On Windows NT, it also contains information about security and hardware.

HKEY_USERS

Available in Windows NT, Windows 95 and Windows 98. This root contains each user's particular settings and application configuration that pertains to individual, users, such as the last documents viewed in an application, window placement or color.

HKEY_CURRENT_CONFIG

Available in Windows 95 and Windows 98 only. It is mapped to a key in HKEY_LOCAL_ MACHINE. This root holds non-user specific configuration information that pertains to hardware, such as printer settings and “Plug and Play” information.

HKEY_DYN_DATA

Available in Windows 95 and Windows 98 only. It contains dynamic information, such as hardware status. The values in this root are of a non-standard format and cannot be seen or edited. (The REGEDIT program that accompanies Windows 95 and Windows 98 will show the value, but does not allow editing.)

Useful Locations in the Registry

HKEY_CURRENT_USER\Software\(CompanyName)\(Program Name)

Utilize this area for configuration information that is specific to an individual user. Simple information (such as color and placement of main windows) can be stored as values; more complex information (such as the last four files viewed) can be stored as new keys, but you can create whatever structure you require. Look at how the Microsoft programs store information here for some ideas (note that even Microsoft does not follow any one standard).

HKEY_LOCAL_MACHINE\SOFTWARE\(Company Name)\(Program Name)

Utilize this area for configuration information that is specific to an individual computer.

HKEY_LOCAL_MACHINE\SOFTWARE\(Company Name)\(Program Name)\Current Version

Version information is located in this key. Some of the standardized value names that should be used are shown below:

Value Name	Description
Description	A text description of the software.
InstallDate	A long integer holding the date.
RegisteredOwner	The registered owner of the software.
MajorVersion	A string containing the most significant version number.
MinorVersion	A string containing the least significant version number.
ServiceName	A short human-readable name.
SoftwareType	The type of program (i.e. “driver”, “service”, “application”,...).
Title	The title of the program.

Any other version information should also be placed here.

HKEY_CLASSES_ROOT\(.ext) = (Program ID) [1]

HKEY_CLASSES_ROOT\(Program ID) = *User readable program description*

Here is where you associate a file type with an executable. First, you associate the file type's extension to a program identifier, which is then given an user-readable description. This description should contain information like the company name, program name and purpose of the file type. Next, tell the Windows shell how to launch a file with this extension:

HKEY_CLASSES_ROOT\(Program ID)\shell\open = "Open"

HKEY_CLASSES_ROOT\(Program ID)\shell\open\command = *executable with full path [flags]* %1[2]

Other verbs can be entered here as well -- just replace the “open” in the previous key with another word, such as “print”. The list of verbs you enter here is seen when a user right-clicks on a file in the Windows 95 or Windows 98 shell. (There is one special verb labeled "PrintTo”. This verb is called when the user drags the file to the printer icon.) You can also specify a Dynamic Data Exchange (DDE) command in place of the command line.

HKEY_CLASSES_ROOT\(Program ID)\shell\open\ddeexec = *DDE command string*

HKEY_CLASSES_ROOT\(Program ID)\shell\open\ddeexec\ Application = *DDE Application name*

HKEY_CLASSES_ROOT\(Program ID)\shell\open\ddeexec\ Topic = *DDE topic name*

You can define what order the verbs are in by entering the verbs into the \shell key in order:

---

[1] That is, the default value is set to (Program ID).

[2] The “%1” is a placeholder for the file for which the program is to be launched.

HKEY_CLASSES_ROOT\(Program ID)\shell *[= default Verb1 [,Verb2 [,..]]*

HKEY_CLASSES_ROOT\(.ext)\ShellNew

One of the features of the Windows 95 or Windows 98 shell is the ability to create new files of a certain type without launching the application. You can add your file type to the list of new file possibilities by including this subkey and setting the proper value in it. The following table shows which value name and value data you should enter to correctly make the new file:

Value Name	Value	Result
Nullfile	""	Creates a file of this type as an empty file.
Data	Binary Data	Creates a new file containing the binary data.
Filename	filename with path	Creates a new file by copying the specified file.
Command	command	Executes the command. Use this to run your own application to create a new file.

HKEY_CLASSES_ROOT\(Program ID)\DefaultIcon

For use with the Windows 95 or Windows 98 shell only. Set this key's default value to the .BMP or .ICO file that contains the icon you want associated with files of this type. You can also extract the icon from an executable's resource -- just include the file name of the program, a comma, and the number denoting the position of the icon within the resource (this is usually 1).

HKEY_CLASSES_ROOT\(Program ID)\CurVer = (ProductName).(number).

A version number can be put here if you are in the Windows 3.1 environment.

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\(Application name)

For use with the Windows 95 or Windows 98 shell only. Set a value named "DisplayName" to the product name, and set a value named "UninstallString" to the filename of the program the shell should use to uninstall your application (including full path and any command line options).

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\SharedDLLs

For use with the Windows 95 or Windows 98 shell only. In this key are the names and full paths of all .DLLs used by installed applications with reference counts of how many applications use each one. This is the method employed by Windows 95 and Windows 98 to decide if it can delete components of uninstalled programs. Make sure your install and uninstall programs correctly update this area.

HKEY_LOCAL_MACHINES\SOFTWARE\Microsoft\Windows\CurrentVersion\AppPaths

For use with the Windows 95 or Windows 98 shell only. This area is used to specify directories for applications and support files. Add a key named after your application here. Set the default value to the path where your .EXE files is located. If you need to, create a value named Path and set it to the path where your .DLL's are located (if not in the same location as your .EXE or \Windows\System). The Windows shell automatically updates this area if you move the executable.

When creating a future update, you might not wish to overwrite the previous Registry entry. Doing so might render the previous version of your software unusable. You can overcome this problem by employing keys to set up different areas with keys for different versions of your software. For example, you could have a:

HKEY_LOCAL_MACHINE\Software\(Company Name)\(Program Name)\1.0

and

HKEY_LOCAL_MACHINE\Software\(Company Name)\(Program Name)\2.0

key to store configuration information for both programs safely.

Use the Registry tools that accompany Windows or the REGBRSR program that accompanies this product to do your own exploring.

Value Data Types

Available in all versions of Windows:

Name	No.	Description
Reg_SZ	1	Null terminated string.

Available in Windows NT, Windows 95 and Windows 98 only:

Name	No.	Description
Reg_Binary	3	Binary data.
Reg_Dword	4	A 32 bit number.
Reg_Dword_Big_Endian	5	A 32 bit number where the most significant byte of a word is in the low-order byte.
Reg_Expand_SZ	2	String that contains references to environment variables (for example, "%PATH%" expands to the current path on the current drive).
Reg_Link	6	Unicode symbolic link
Reg_Multi_SZ	7	Array of strings, terminated by two nulls.
Reg_None	0	No defined value type.
Reg_Resource_List	8	List of hardware resources for device-drivers. Each element is a Reg_Full_Resource_Descriptor.
Reg_Full_Resource_Descriptor	9	Description of hardware resources a device is using.
Reg_Resource_Requirements_List	10	List of possible hardware resources a device can use.

The Registry Component and ActiveX Controls

Unfortunately for Visual Basic users, the Registry API functions are primarily designed for C/C++ programmers. Value data types don't match Visual Basic data types, and many of the functions require pointers. The Desaware Registry Component (dwReg.dll) and the Registry Controls (dwReg16.ocx and dwReg32.ocx) provide an easy way to browse and modify the Registry and Registration Database. It provides all the functionality of the Registry API functions and more -- designed for the Visual Basic programmer.

This control takes the approach of treating the registry as if it were a simple file system, and provides a mechanism for working with the registry that is familiar to Visual Basic programmers as it is similar on the standard Visual Basic file commands.

Registry Component Example

Here is a brief example of how you could use the Registry component to store the version information of an imaginary program. First, add the component to your project. This is done by checking the "Desaware Registry 2.0 Component" line in the Visual Basic References dialog box. The following line creates an instance of the control in your code.

```
Dim Registry1 as New dwRegistry
```

Confirm that the control is set to the correct tree. Version information belongs in HKEY_CLASSES_ROOT.

```
Registry1.CurrentRoot = Hkey_Current_User
```

Next, verify that a key containing version information already exists by trying to set the CurrentKey to it.

```
On Error GoTo notRegistered
Registry1.CurrentKey = "\SOFTWARE\Xcorp\Xapp"
On Error GoTo 0
```

If the key does not exist, create it. Remember that by creating a key, you also automatically create all of the parent keys.

```
notRegistered:
Registry1.CreateKey
   "\SOFTWARE\Xcorp\Xapp\CurrentVersion"
```

Now we set the CurrentKey to that key, and add the values that hold the version information.

```
Dim TmpVar as Variant

Registry1.CurrentKey =
   "\SOFTWARE\Xcorp\Xapp\CurrentVersion"
TmpVar = "XYZapp"
Registry1.SetValue "Title", TmpVar, Reg_SZ, 6
TmpVar = "1"
Registry1.SetValue "MajorVersion", TmpVar,
   Reg_SZ, 1
TmpVar = "0"
Registry1.SetValue "MinorVersion", TmpVar,
   Reg_SZ, 1
```

Registry ActiveX Controls

The dwReg16.ocx and dwReg32.ocx ActiveX controls have all the same methods and properties as the component version. These are used when you are making a 16-bit program or where maximum compatibility between 16 and 32 bit programs is needed. The component version is preferred for 32-bit programs because it does not require the MFC40.DLL and MSVCRT40.DLL files for distribution. The main difference in code between the control and component version is that you do not need to create an instance of the control version – as soon as you place the Registry control on your form, the `Registry1` object exists and can be used. The controls can be also be manipulated in design time, making debugging and experimenting easier.

Property Page for ActiveX Controls

You can access the Registry control's Property Page by clicking on the "..." button in the **(Custom)** property in the Properties Window or by right-clicking on the control. This will bring up a tabbed dialog that depicts the editable properties of the Registry OLE control. The only tab is called **Current Key**. You can modify the current state of the control by changing the information in this dialog and pressing **Apply** or **OK**. You can change the currently accessed key by typing the new key (with full path) into the **Current Key** textbox. Changing the text in the **Current Value** textbox will change the value for the current key. The list of possible root keys is available in the **Current Root** combobox. If you are using Windows NT or Windows 95, you will see HKEY_CURRENT_CONFIG and HKEY_DYN_DATA, which exist in Windows 95 only. Attempting to select these will result in a message box reporting an error, and the root will revert to the previous selection. The number of subkeys of the current key can be found in the read-only textbox labeled **Number of Subkeys.**

The number of the values contained in the current key are found in the read-only **Number of Values** textbox. The current state of the KeyLock property can be seen in the **Keep Registry Open** checkbox.

Registry Properties

Registry - CurrentKey

This property contains the current key upon which other properties and the majority of the methods are based. It starts at the root of the **CurrentRoot**. To change this to a different key, set **CurrentKey** with a full path (list the parent keys all the way to the root as necessary). If you want to change the key relative to the current key, use the **ChangeKey** method. If **CurrentKey** is set to a non-existent or non-accessible key, it is reset to the root of the **CurrentRoot** and an error is generated. Err.Description is "Current Key Error" and Err.Number depends upon which error caused the problem.

Registry - CurrentRoot

This specifies in which root of the Registry the **CurrentKey** currently resides. The possible values depend upon which operating system is being used. Windows 3.1 only has **HKey_Classes_Root**. Windows NT has three more. Windows 95/98 has the same ones as Windows NT in addition to two more. However, the controls will display all root keys contained in Windows 95/98. If you select a key that does not exist in the operating system that you are using, an error will result. If the root is changed, **CurrentKey** is changed to become the root of the new root. All operations take place in the current root only. Any problem will generate a "Invalid Root Value" error.

Registry - DefaultValue

This holds the "default" value, the value directly associated with the **CurrentKey**. It is the only type of value available in Windows 3.1. This is the value known as "<no name>" in Windows NT and "(default)" in Windows 95/98 registry editors. If the **CurrentKey** does not have a value, or if the default value is not a string, **DefaultValue** is empty. When **DefaultValue** is changed, the Registry is updated to reflect the change. If the default value exists, it is also represented as a value in the **ValueNameArray** (with an empty string as its name). **DefaultValue** is not very useful in roots other than **HKey_Classes_Root**, as this property only holds **Reg_SZ** or **Reg_Expand_SZ** values.

Registry - FindResultKey

This property holds the latest key found with **FindFirstKey**, **FindNextKey**, **FindFirstValue**, **FindNextValue**, **FindFirstValueName** or **FindNextValueName**. This is a read-only property. It is empty by default.

Registry - FindResultValueName

This is a name of a value. It holds the latest result in a search for a certain value name, or it specifies the latest value found with **FindFirstValue** or **FindNextValue**. If a search was made by **FindFirstKey** or **FindNextKey**, **FindResultValueName** will be blank. This is a read only property. To find out more about the value named here, use the various GetValue methods.

Registry - KeyLock

When **KeyLock** is set to True, the **CurrentKey** is kept open. This speeds up access to the Registry, but it is somewhat less safe. The increases in speed take effect during those functions that occur at the **CurrentKey** (such as **CreateKey** or **DeleteKey**) but not during those functions that do not occur at the **CurrentKey** (such as the Find functions). You should probably set **KeyLock** to True just before a series of calls to read or change the Registry, then set **KeyLock** back to False. Never leave this True longer than needed. **KeyLock** is False by default.

Registry - NumOfSubkeys

This holds the number of subkeys currently in the **CurrentKey.** Use **SubkeyArray()** to access them.

Registry - NumOfValues

This holds the number of values currently in the **CurrentKey.** Use the **ValueNameArray()** property to access the names of each of the values, then the various GetValue methods to obtain the actual data in the value. In Windows 3.1, this is always either 0 or 1.

Registry - SubkeyArray (Index as Integer)

An array of all subkeys of the **CurrentKey** is kept in the **SubkeyArray()**. The array starts at 0. The number of subkeys is in the **NumOfSubkeys** property. If the index is beyond either of these bounds, **SubkeyArray()** returns a null string and a standard Out Of Bounds error.

Registry - ValueNameArray (Index as Integer)

An array of the names of all the values held in the **CurrentKey** are kept in the **ValueNameArray()**. The array starts at 0. The number of values is in the **NumOfValues** property. If the index is beyond either of these bounds, **ValueNameArray()** returns a null and an Out Of Bounds error. This is not available on the Properties palette. A blank **ValueNameArray** element actually holds the "name" of the default value (blank because a default value does not have a name).

Registry Methods

Registry - ChangeKey (ChangeStr as String)

ChangeKey allows you to change the **CurrentKey** using syntax similar to changing a file directory. If ChangeStr begins with a backslash ("\"), the change is relative to the root key -- otherwise, it is relative to the **CurrentKey**. A dot-dot ("..") is used to move from a subkey to its parent. If **ChangeKey** cannot get to the specified key, it causes a "Trying to change to invalid key" error. You cannot go to the parent of the root key, even if such a key exists.

For example, you cannot use **ChangeKey**("..") to move from the root key of **HKey_Current_User** to **HKey_Users** even though **HKey_Current_User** is technically a subkey of **HKey_Users**.

Registry - CreateKey (NewKey as String)

This function is used to make a new key. The NewKey string can specify an absolute path, or one relative to **CurrentKey**. Any subkeys of the new key that do not exist are also created. If the key specified already exists, nothing happens. If a new key cannot be made, **CreateKey** causes a "Failure trying to create a subkey" error.

Registry - DeleteKey (Key as String, EraseSubkeys as Boolean)

This function is used to delete an existing key. The key may be specified relative to the **CurrentKey** (without a leading "\") or as an absolute path (with leading "\"). If EraseSubkeys is True, any subkeys are also deleted, otherwise if the key has any subkeys it is left alone. If the key cannot be deleted for some reason, **DeleteKey** generates the "Failure trying to delete key" error.

Registry - DeleteValue (Name as String)

This function is used to delete an existing value. The value to be deleted must reside in the **CurrentKey**. If Name is blank, then the default value is erased. If the value cannot be deleted successfully, **DeleteValue** sends a "Failure trying to delete value" error.

Registry - EnableComponent (LicenseKey as String)

This method allows you to use the Registry component in your own Visual Basic components, such as ActiveX controls or ActiveX documents.

You first create a key by using the dwLicGen.exe program. This program will ask you to enter the compiled name of your component (not including a path, but including an extension). Each key is unique to a component name. If you change your component name, you'll have to create a new license key. After the program generates the key, you will have the option to copy it to the clipboard to post into your application.

You then call the **EnableComponent** method before you access any of the components methods or properties. It is best placed in your control's Initialize or Sub Main functions. If you ever unload or remove the last reference to the Registry component, you will have to go through this process again upon creating a new reference. This method is only available in the dwReg.dll component – you cannot use the dwReg32.ocx control in your own ActiveX controls.

```
Dim DwReg As New dwRegistry

Private Sub UserControl_Initialize()
   On Error GoTo initfailed
   ' Your key will be different.
   DwReg.EnableComponent "2E3499248E322D3636853B53"
   Exit Sub
initfailed:
   ' Do appropriate error handling here.
End Sub
```

Registry - FindFirstKey (Name as String, Case as Boolean, FullString as Boolean, StartAtRoot as Boolean)

Call this function first in order to search for any keys that have a name matching Name. If Case is True, than the comparison between Name and the keys are case sensitive. If FullString is True, then the two strings are compared over their whole length. If FullString is False, then Name can be a substring of a key. If StartAtRoot is True, then the search begins at the root key and progresses through all subkeys -- otherwise, the search starts at the **CurrentKey** and progresses through its subkeys. If at least one match was found, **FindFirstKey** will return True and **FindResultKey** will hold the first key found. If no matches were found, **FindFirstKey** will return False. This function might take several seconds to complete, depending upon the size of the root being searched.

Registry - FindFirstValue (Value as Variant, TypeOfValue as Long, SizeOfValue as Long, FullData as Boolean, StartAtRoot as Boolean)

Call this function first in order to search for any keys that have a value matching Value. The Registry type of the variant Value should be specified in **TypeOfValue**. The size of the Value (in bytes) should be in **SizeOfValue**.

If FullData is True, then the two values are compared over their whole length. If FullData is False, then Value can be located anywhere within the value being viewed. (FullData is ignored if TypeOfValue is **Reg_Dword**). If StartAtRoot is True, then the search begins at the root key and progresses through all subkeys -- otherwise, the search starts at the **CurrentKey** and progresses through its subkeys.

If Value is a string (**REG_SZ**, **REG_EXPAND_SZ** or **REG_MULTI_SZ**), any comparisons are made without regard to case.

If at least one match was found, **FindFirstValue** will return True and **FindResultKey** and **FindResultValueName** will hold the first key found. If no matches were found, **FindFirstValue** will return False. This function may take several seconds to complete, depending upon the size of the root being searched.

Registry - FindFirstValueName (Name as String, Case as Boolean, FullString as Boolean, StartAtRoot as Boolean)

Call this function first in order to search for any keys that have a value named Name.

If Case is True, than the comparisons between Name and the value name are case sensitive. If FullString is True, then the two strings are compared over their whole length. If FullString is False, then Name can be a substring of a value name. If StartAtRoot is True, then the search begins at the root key and progresses through all subkeys -- otherwise, the search starts at the **CurrentKey** and progresses through its subkeys.

If at least one match was found, **FindFirstValueName** will return True and **FindResultKey** and **FindResultValueName** will hold the first key and value name found. If no matches were found, **FindFirstValueName** will return False. This function does not exist in Windows 3.1. This function may take up to ten seconds to finish, depending on the size of the root being searched.

Registry - FindNextKey ()

After the first key is found using **FindFirstKey**, use this function to obtain the remainder of the keys matching those search parameters. If there are more keys that match, each call to **FindNextKey** will put another key into the **FindResultKey** property. If there are no more matching keys, **FindNextKey** returns False and the **FindResultKey** property is cleared.

Registry - FindNextValue ()

After the first key is found using **FindFirstValue**, use this function to obtain the remainder of the keys with values matching those search parameters. If there are more keys that match, each call to **FindNextValue** will put another key into the **FindResultKey** and **FindResultValueName** properties. If there are no more matching values, **FindNextValue** returns False and FindResult properties are cleared.

Registry - FindNextValueName ()

After the first key is found using **FindFirstValueName**, use this function to obtain the remainder of the keys with value names matching those search parameters. If there are more keys that match, each call to **FindNextValue** will put another key into the **FindResultKey** and **FindResultValueName** properties. If there are no more matching keys, **FindNextValueName** returns False and FindResult properties are cleared. This function does not exist in Windows 3.1.

Registry - FlushRegistry ()

In 32 bit versions of Windows, much of the Registry is kept in memory to improve performance. Changes to the Registry are made to the copy in memory. The actual Registry on disk is usually updated a few seconds later. Call this function when you want to be 110% sure the Registry is completely saved to disk. This call is time intensive, so use it sparingly, if at all. **KeyLock** should be False before calling this function. **FlushRegistry** has no effect in Windows 3.1, because its Registry is always immediately updated to disk. If the Registry cannot be flushed, **FlushRegistry** returns False.

Registry - GetValueData (Name as String, AsString as Boolean)

This function returns the data in the value named Name residing in **CurrentKey**. Use the **GetValueSize** and **GetValueType** functions to correctly set aside the right amount of memory for the variant. See the *Data Type* section of this manual for more information on how the data is actually stored. If the **AsString** parameter is set to True, then any binary data will be returned in the form of a string. If it is set to False, then any binary data will be put into an array of bytes. If the specified value does not exist, **GetValueData** returns a Null.

Registry - GetValueSize (Name as String)

This function returns the size, in bytes, of the data in the value named Name residing in **CurrentKey**. If the value specified does not exist, **GetValueSize** returns 0.

Registry - GetValueType (Name as String)

This function returns a number representing the type of the data held in the value named Name in **CurrentKey**. If the value specified does not exist, **GetValueType** returns 0.

Registry - SetValue (Name as String, Data as Variant, TypeOfValue as Long, SizeOfValue as Long)

This function is used to modify the contents of a value in the **CurrentKey**. If the value named Name does not exist, it is created. If an error occurred while attempting to set the value, **SetValue** generates the error code "Cannot Create\Change Value". Note that the "Cvar" Visual Basic function is not a reliable way of converting a variable to the Variant type.

Conversion of Value Data Types for Visual Basic

Values of type Reg_SZ, Reg_Expand_SZ, or Reg_Multi_SZ are treated as Visual Basic strings.

Values of type Reg_Dword or Reg_Dword_Big_Endian are treated as Visual Basic long integers.

Other values are treated as binary information. These can be held in either Visual Basic strings or byte arrays.

Reg_Resource_List, Reg_Full_Resource_Descriptor, and Reg_Resource_Requirements_List are utilized by device drivers and other very low level Windows elements. Do not modify these unless you really know what you are doing.

Possible Error Codes

Name	Description
Reg_E_Access_Denied	You do not have the authority to access the key or value.
Reg_E_BadDB	The Registry is corrupt. You have to either manually repair it, restore a backup, or reinstall Windows.
Reg_E_BadKey	The particular key is corrupted or does not exist.
Reg_E_BadRoot	The value does not correspond to any existing root. Remember that some roots only exist in certain operating systems.
Reg_E_CantOpen	The key cannot be accessed.
Reg_E_CantRead	The key cannot be read. In Windows NT, this might indicate that you do not have permission to read that particular key or value.
Reg_E_CantWrite	The key cannot be changed. In Windows NT, this might indicate that you do not have permission to change the particular key or value.
Reg_E_Child_Must_Be_Volitile	You cannot make a subkey for this key because the key might change at anytime.

Reg_E_Key_Deleted	The key is in the process of being deleted. You cannot change it or any of its subkeys or values.
Reg_E_Key_Not_Found	The registry changed after CurrentKey had been set.
Reg_E_Local_Machine_Root	The root structure of the **Reg_Local_Machine** root cannot be changed.
Reg_E_No_Log_Space	There is not enough disk space to save the Registry.
Reg_E_Path_Not_Found	The path of the key you specified does not exist.
Reg_E_Registry_Corrupt	The Registry is corrupt. You have to either manually repair it, restore a backup, or reinstall Windows.
Reg_E_Registry_IO_Failed	The Registry cannot be accessed. The disk might be full.

Registry Browser

To aid managing the Registry, StorageTools includes a program called the **Registry Browser** (RegBrs32.exe or RegBrs16.exe). It allows complete access to the Registry, letting you view, edit, create or delete values and keys. It also includes powerful searching abilities that the standard Registry editors do not have. The complete Visual Basic source code is included, and it may be incorporated into your own projects.

Figure 5 depicts the Main Form for the Registry Browser.

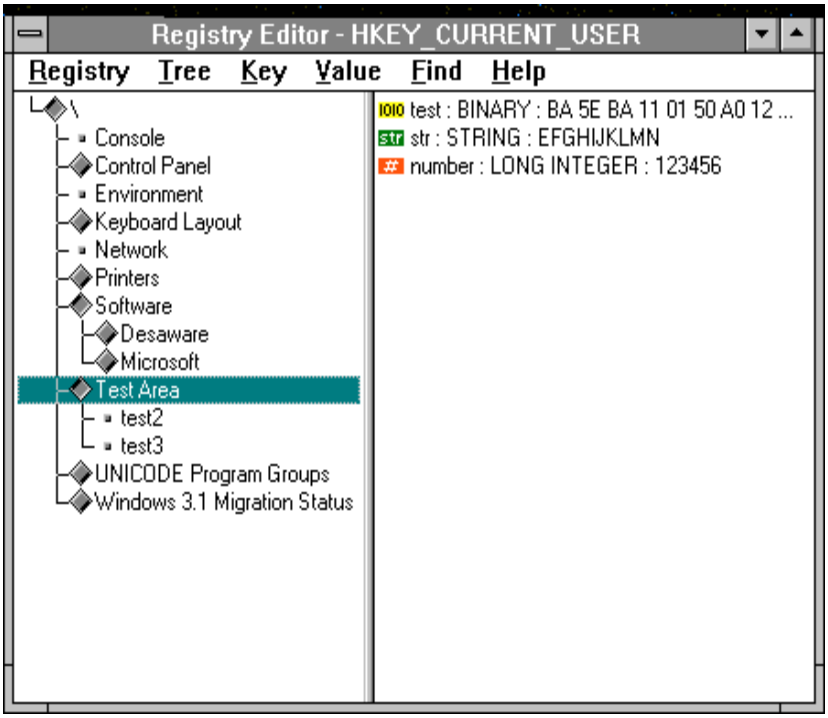

Figure 5
Main Form Registry Browser

The currently open root is shown in the title bar. The window on the left presents a hierarchical list of the keys, much like the list of directories in a File Manager window. Keys that contain more keys are preceded by a diamond icon. You can show or hide subkeys by double-clicking on the parent key. The window on the right shows the values in the currently highlighted key. Each value is preceded by an icon to help assist in the recognition of its type. The name, type and value data are separated by colons. Only the first 16 bytes of binary data are shown.

Registry Menu

The only item besides **Quit** is **Force Changes to Disk**. Occasionally, changes made to the Registry are buffered, and might take a while before being saved to disk. This item will guarantee that all modifications have been permanently stored.

Root Menu

These menu items list all of the available root keys. Because there is only one root key under Windows 3.1, the root menu will not exist when this program is run there.

Key Menu

All of these apply to the currently selected key in the left window. **Add New Key** brings up a dialog box that allows you to type the name of a new key to be placed within the currently selected key. **Delete Key** erases the current key and all keys and values contained within it. **Refresh Key** rereads all of the values and subkeys of the currently selected key in order to adjust for any changes other programs might have made in the Registry.

Value Menu

Add New Value brings up a dialog box that allows you to select the name and type of the value. After that is accomplished, another dialog box appears allows you to edit the contents of the value (see the "**Data Editing Dialogs**" section later in this chapter). **Delete Value** erases the currently selected value. **Edit Value** brings up a dialog box that lets you edit the value data. Double-clicking on a value also brings up an edit dialog box.

Find Menu

You can search for particular keys, value names and value data. Figure 6 illustrates the dialog box associated with the **Find Key.**

Figure 6
Find Key Dialog Box

Input the key name (or portion thereof) into the text box at the top. If you want to find key names that contain a fragment of a name, make sure **Match Whole Key Only** is not checked. A more specific search (**Match Case** is checked, **Match Whole Key Only** is checked, or **Start Search at Current Key** is checked) takes less time.

The **Find Value Name** dialog (which works exactly like the **Find Key** dialog).

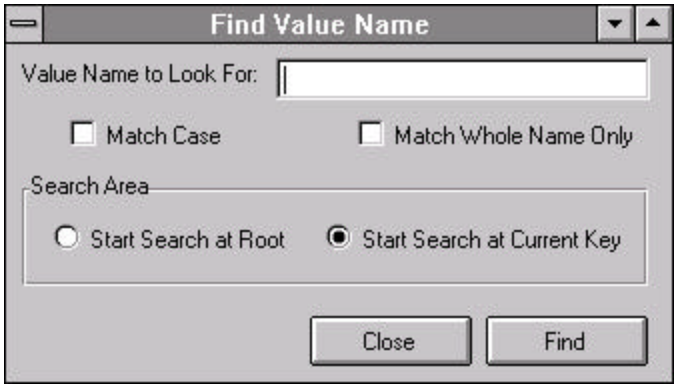

Figure 7
Find Value Name Dialog Box

The Find Value Data dialog is pictured below (Figure 8):

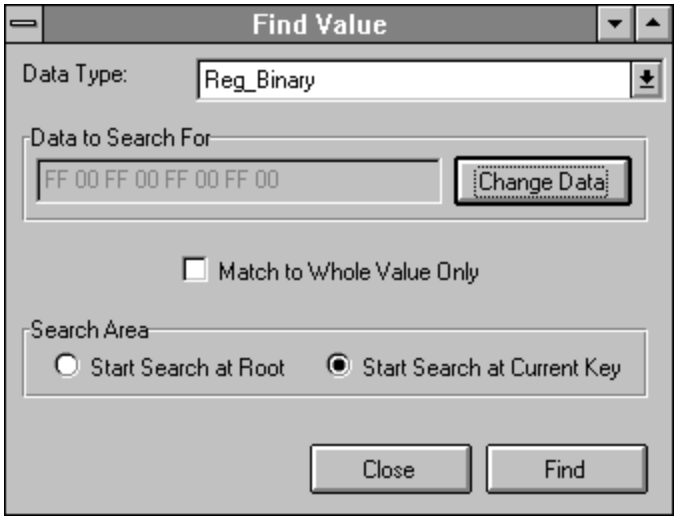

Figure 8
Find Value Data Dialog Box

First, select the type of data for which you are searching. Then click the Change Data button to input the data that is being sought. This button will bring up a dialog that lets you properly enter the data. When you are done, a brief portion of the data you are looking for will be visible to the left of the button. The check boxes and radio buttons are the same as those in the other find routines except that **Match Case** is missing (any string comparisons are done in a case insensitive manner).

Help Menu

This menu lets you access the about box.

Data Editing Dialogs

Because values can contain up to ten different types of data, there are four different dialog boxes for editing data.

The **Change String Data** dialog is brought up when Reg_SZ or Reg_Expand_SZ values are to be edited. Simply type the new information into the textbox.

The **Change Long Integer Data** dialog is similar. It is used to edit values of the types Reg_Dword_Big_Endian and Reg_Dword. Only numbers can be typed into the textbox. If the number typed is too large, an error will result and you will be given another opportunity to input the number.

The **Change Multi-Line String Data** dialog box is for values of type Reg_Multi_SZ. It has a large, multi-line textbox, but is otherwise like the **Change String Data** dialog.

Finally, the **Change Binary Data** dialog box lets you manipulate values of any other type. The binary data is shown as hexadecimal digits, and no characters besides 0 through 9 and A, B, C, D, E and F will be accepted. A ruler across the top of the edit box shows you how many bytes of data exist.

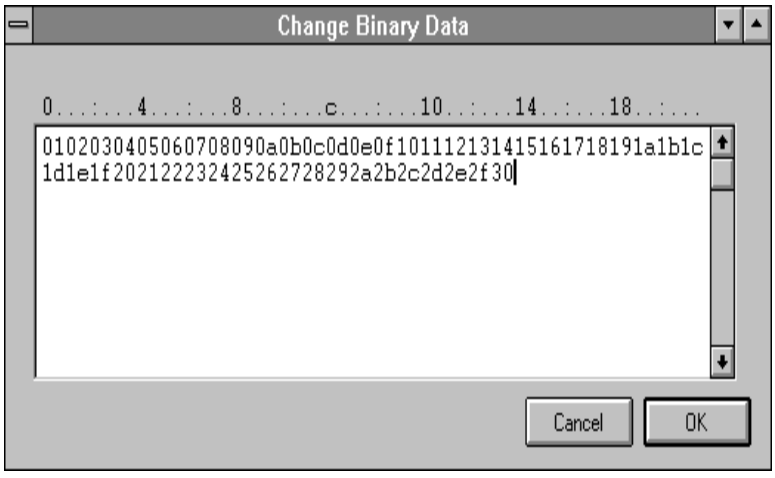

Figure 9
Change Binary Data Dialog Box

WARNING!

Remember that the Registry holds a great deal of information that both Windows itself and many Windows applications depend upon. Be careful when making changes to any part of the Registry that you did not create. A corrupted Registry might render a program or Windows itself unusable.

The Desaware Resource Compiler

One common problem that programmers face is the need to include constant data in a program. This is most typically text data or image data. Ancient versions of Basic supported the 'Data' statement that allowed you to place the data directly in your program code. This command has never been part of Visual Basic.

So what options are available for embedding data into your executable files? You can, of course, take the simple code approach. Perhaps something like this:

```
dim EmbeddedStrings(3)
EmbeddedStrings(0) = "first string"
EmbeddedStrings(1) = "second string"
EmbeddedStrings(2) = "third string"
EmbeddedStrings(3) = "fourth string"
```

This approach works nicely, but does have disadvantages. It has an impact on performance, since the code that initializes the variables takes time to execute. Also, if you wish your program to support multiple languages, you will need to include strings for each language in your code.

Windows addresses these problems by making it possible to define a part of your executable file as containing fixed data called resources. A number of predefined resource types are defined, of which the most interesting ones to Visual Basic programmers are strings, bitmaps, icons and user defined binary data. Visual Basic allows you to access resources in your file using the LoadResData, LoadResPicture and LoadResString functions. You can also use API resource functions to access resources in other programs or dynamic link libraries.

Visual Basic allows you to embed a resource file into your application by adding a resource file (.res extension) to your project. The enterprise edition of Visual Basic includes a resource compiler, as does Visual C++. This resource compiler compiles a resource command file (.rc extension) into a .RES file. The .RC file is a text file that lists resources and is moderately cryptic to work with. A more complete introduction to resources, why they exist and how they can be used can be found in both the "Visual Basic Programmer's Guide to the Win32 API" and the original "Visual Basic Programmer's Guide to the Windows API".

Suffice to say that in order to use resource files with Visual Basic you must first obtain a resource compiler, write an RC file, and compile that RC file any time any of the resources change. This approach works, but is rather awkward. The Desaware resource compiler was designed to fill this need in an efficient and easy to use fashion.

Not only that, but following our tradition of offering extensive sample code, the complete source code for the resource compiler is included. Not only will you see how resource files are organized and how they are created, you'll also see an example of a Visual Basic add-in, and an example of one way to use the Desaware Storage Control and OLE structured storage.

Final Comments - Customer Support

For information on customer support and last minute changes, refer to the file readme.wri on the StorageTools CD. This file is compatible with write.exe (included with each copy of Windows).

There is a saying in the software world that no non-trivial program is completely bug free. The corollary to that saying is that no program with more than ten lines in it is non-trivial. StorageTools is emphatically non-trivial....

StorageTools has undergone extensive testing to make it as bug free as possible. Nevertheless, it is possible that some have crept through. Please write or send us a fax if you find one, and include all of the steps that you needed to go through to reproduce the problem. Also, if there are any files needed to reproduce the error, send them to us on a diskette.

Our web page has a technical support area where frequently asked questions are posted. Please feel welcome to visit this site to see if your question has already been answered.

We would also appreciate your suggestions regarding this manual. Specific comments and questions are especially welcome. We have attempted to address as many questions as possible, but if you run into something confusing, please let us know so that we can incorporate revisions into the next edition.

Finally, and perhaps most important, we would love to hear your suggestions for improvements to StorageTools, or any suggestions you may have for new products or custom controls.

Please address all correspondence to:

Desaware, Inc.
3510 Charter Park Drive, Suite 48
San Jose, CA 95136
Phone: 408/404-4760, Fax: 408/404-4780
Web Site: http://www.desaware.com
Email: support@desaware.com

Other Sources of Information

Here are several other resources that we recommend for advance Windows development.

Regular Expressions with .NET

This ebook is intended to be a complete introduction to Regular Expressions that can even be read and understood by programmers who have never heard of them. It is also intended to help experienced Regular Expression programmers come up to speed quickly on the .NET implementation of Regular Expressions.

This ebook is available from Amazon.com.

Visual Basic.NET or C# ... Which to Choose?

In this ebook you will find an in-depth comparison of the two languages. In a feature-by-feature, head-to-head contest, Dan pulls no punches in calling the winner in each case.

But a technical comparison is only the beginning. With a keen eye for the business issues involved in language choice, the author focuses on the economic issues involved in this decision, considering the cost of retraining and long-term support, as well as that of initial development.

This ebook is available from Amazon.com.

Introduction to NT/2000 Security Programming with Visual Basic

NT Security is a subject that is intimidating, to say the least. But if you dig past the confusing acronyms, you'll find that it's actually very easy to understand. This article, based on Dan Appleman's well received talks at VBits, will help you get started on the right foot with NT security, and give you the foundation of knowledge you'll need to understand even the most obscure security concepts. It will also introduce you to techniques for adding security based features to your applications (with an emphasis on Visual Basic applications).

This ebook is available from Amazon.com.

Moving to VB.Net:Strategies, Concepts and Code

Written by Daniel Appleman (president of Desaware) and published by Apress (ISBN 1893115976).

VB.Net is not Visual Basic. Porting is stupid. COM is "dead". These are just a few of the things you'll learn as Dan takes you on a journey unlike any other into the world of VB.Net. Covers adoption strategies, unlearning VB6 concepts that are fatal in VB.Net, and analysis of language changes that goes beyond the documentation.

Available at most good bookstores, or directly from Desaware at a 20% discount - call (408) 404-4760 or email support@desaware.com.

Dan Appleman's Visual Basic Programmer's Guide To The Win32 API

Written by Daniel Appleman (president of Desaware) and published by McMillan, (ISBN 0-672-31590-4) - this sequel to the original 16 bit API Guide applies the same philosophy to teaching the Win32 API to developers using Visual Basic and VBA based applications. With more examples, more functions, more tutorial style explanations and a full text searchable electronic edition on CD-ROM, this book should prove a worthy successor to the 16 bit API book. Covers Visual Basic version 4 through 6.

Available at most good bookstores, or directly from Desaware at a 20% discount - call (408) 404-4760 or email support@desaware.com.

An upgrade CD is available for owners of the “PC Magazine's Visual Basic Programmer's Guide to the Win32 API” ISBN: 1-56276-287-7 for $24.99 + s&h directly from Desaware. Refer to our web site at www.desaware.com for additional information.

Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed

Written by Daniel Appleman (president of Desaware) and published by McMillan, (ISBN 1-56276-576-0) - this book is designed for those programmers interested in using Visual Basic's object oriented technology to develop ActiveX components including EXE and DLL servers, ActiveX controls and ActiveX documents. Unlike many books that simply rehash the Visual Basic documentation, this one serves as a commentary to clarify and extend the documentation. Of special

interest to VersionStamper customers will be the chapters on OLE and COM technology that will help them further understand the process of registering components, and the chapters on versioning and licensing.

The VB6 version also includes two new chapters on IIS Application development.

Available at most good bookstores, or directly from Desaware at a 20% discount - call (408) 404-4760 or email support@desaware.com.

Dan Appleman's Win32 API Puzzle Book and Tutorial for Visual Basic Programmers

Written by Daniel Appleman (president of Desaware) and published by Apress (ISBN# 1-893115-01-1). Appleman's Win32 API Guide covers 700 API functions. This book covers the other 7800. How? By teaching you everything you need to know to read and understand the Microsoft C documentation and create correct API declarations for use in Visual Basic. Presented in an entertaining puzzle/solution format that challenges you to solve real world API problems. In depth tutorials take you behind the scenes to understand what really happens when you call an API function from VB.

Available at most good bookstores, or directly from Desaware at a 20% discount - call (408) 404-4760 or email support@desaware.com.

How Computer Programming Works

A useful book for future programmers or anyone interested in explaining important computer programming concepts. Full color illustrations help to visually explain important topics! New expanded section on computer programming for the Internet.

Just as a child must learn the alphabet before they can read, future programmers must understand certain concepts before they can write their first program. This unique book uses full color illustrations to help the reader to truly understand the underlying computer science on which all programming is based. ISBN 1-893115-23-2.

Available at most good bookstores, or directly from Desaware at a 20% discount - call (408) 404-4760 or email support@desaware.com.

The Desaware Visual Basic Bulletin

and other related technical articles. At the Desaware website: http://www.desaware.com.

PC Magazine's Visual Basic Programmer's Guide To The Windows API

Written by Daniel Appleman (president of Desaware) this book is intended to help Visual Basic programmers navigate the complexities of Windows. It is the only text on Windows that is designed specifically for Visual Basic programmers, and the only one that covers the interactions between Visual Basic and Windows.

Available on CD Rom only from Desaware. Call (408) 404-4760 or email support@desaware.com.

Msdn.microsoft.com

This web site is a comprehensive reference.

Desaware Product Descriptions

Thank you for your purchase of this Desaware product. We have additional quality software to enhance your programming efforts. Please visit our web site at www.desaware.com for detailed descriptions and product demos.

SPYWORKS Standard 6/Professional 7.0

SpyWorks in a nutshell? Impossible!

You're going to want to download the SpyWorks demo to even begin to understand its capabilities. This product has been evolving for several years, and it includes so many features it's hard to know where to begin. SpyWorks is a VB power tool. When you need to override VB's default behavior or to extend VB's functionality, you will want to use SpyWorks.

Do *That* in Visual Basic??

Want to put VB to the test? Want to learn advanced programming techniques? Want to keep the productivity of VB and have the functionality of C++? SpyWorks contains the low level tools that you need to take full advantage of Windows. Here are just a few of the features of this multi-faceted software package. For instance, have you ever wanted to detect keystrokes on a system-wide basis or detect when an event occurs in another application or thread using subclassing or hooks? SpyWorks can help you solve these problems by letting you tap into the full power of the Windows API without having to be an expert. SpyWorks lets you export functions from VB DLL's so that you can create function libraries, control panel applets, and NT Services. With its ActiveX extension technology, you can call and implement interfaces that VB5 or 6 do not support. SpyWorks includes the Desaware API Class Library, which assists programmers in taking advantage of the hundreds of functions that are built into the Windows API. SpyWorks is available in either the Professional (Pro) or Standard edition.

The Professional Edition includes .NET support for keyboard hooks, window hooks and subclassing (including cross-task subclassing) with examples in both Visual Basic.NET and C#. Additionally, a WinSock component with comprehensive VB source code that gives you complete control for Internet/intranet programming.

Other features are the NT Service Toolkit *Light Edition*. This application is a subset of the Desaware NT Service Toolkit product. It allows a developer to create true NT services using Visual Basic. A background thread component that allows you to easily create objects that run in a separate background thread.

It also contains extensive sample code and three product updates.

The Professional Edition includes the Winsock Library, NT Service support and many other additional features & samples, plus three free updates. SpyWorks 2.1 (VBX Edition) is included in the Pro Edition.

SpyWorks Standard is a subset of Professional. A feature comparison is available on our web site.

Supports VB 4, 5 & 6, Windows 95, 98, 2000, NT and ME depending upon which version (or edition) of SpyWorks.

VERSIONSTAMPER 6.5

Distributing Component-Based Applications? Beware DLL HELL!

You've distributed your application and it's working fine. But your end user is still in charge of their system. What happens when they install a program that overwrites a component that your software needs to run? Can you verify that your users have the correct files required by your application? Can you really afford to spend two hours on the phone trying to figure out exactly what went wrong? Now you can easily avoid component incompatibilities by adding VersionStamper to your toolkit. It lets you check the versions of your program's components on your end user’s system, and correct the problem.

You are in control!

DLL Hell is a big problem, and with VersionStamper you can be in control of how this problem is detected and corrected. You determine dependency scanning (file size, date, version or other parameter), how and when the dependency scanning is done (upon start up, at midnight, at user's discretion), and how you want the problem resolved (automatically, an email message to your help desk, from a dependency list on your web site and more). This means you can handle versioning problems as simply as using a message box to call tech support, or even automatically updating the invalid components over the internet or corporate network. Imagine your application updating itself without user (or programmer) intervention! Imagine the hours and money saved in tech support calls! You can even use VersionStamper for incremental updates and bug fixes.

Is This For Real?

No, you don't have to pay a fortune in distribution fees - there are no run-time licensing fees. VersionStamper comes with a great deal of sample code. Don't distribute a component-based application without it!

Checks the versions of your dependent files and notifies you or the user of potential problems.

Internet extensions allow you to update versions across the Internet/intranets.

Cool and USEFUL sample programs show you how it works.

Includes VB source code for the VersionStamper components that you can use in your applications.

NT SERVICE TOOLKIT 2.0

Create a fully featured service in minutes using Visual Basic – even debug your service using the Visual Basic environment! Supports all NT service options and controls. Adheres to all Visual Basic threading rules. Background thread support allows easy waiting on system and synchronization objects. Client requests supported on independent threads for excellent scalability, with client impersonation available allowing services to act on behalf of clients in their own security context. Client requests and service control possible via COM/COM+/DCOM.

Simulation mode for testing as an independent executable. Create control panel applets for service control and other purposes.

DESAWARE EVENT LOG TOOLKIT 1.0

Visual Basic allows you to log events to the NT/2000 event log, but does not allow you to create custom event sources - so every event belongs to the application VB runtime, descriptions are limited, and event categories unavailable. Even if you use the API to log events, creating custom event sources for your application is not supported by VB, and is difficult with C++.

Desaware's new Event Log Toolkit makes creation of event sources easy, and provides all the tools needed to create and log custom events. Now your applications and services can support event logs in a professional manner, as recommended by Microsoft

STORAGETOOLS ver 3.0

StorageTools is your key to the OLE 2.0 Structured Storage Technology. Structured Storage allows you to create files that organize complex data easily in a hierarchical system. It is like having an entire file system in each file. OLE 2.0 takes care of allocating and freeing space within a file, so just as you need not concern yourself with the physical placement of files on disk, you can also disregard the actual location of data in the file. Additionally, with its support for transactioning you can easily implement undo operations and incremental saves in your application. StorageTools allows you to take advantage of the same file storage system used by Microsoft's own applications. In fact, we include programs (with Visual Basic source code) that let you examine the structure of any OLE 2.0 based file so that you can see exactly how they do it!

StorageTools includes documentation and controls to make it easy to work with the registration database under Windows 3.1, Windows NT & Windows 95/98 and 2000. For Visual Basic 4-6. We also include a simple resource compiler (with Visual Basic source code) so that you can create your own .RES files for use with Visual Basic.

StorageTools version 3.0 also includes the Desaware File Property component and .NET support.

StorageTools includes 16 & 32 bit ActiveX/ATL controls, extensive documentation and sample code.

DESAWARE ACTIVEX GALLIMAUFRY Ver. 2

What is it?

gal·li·mau·fry (gàl´e-mô¹frê) noun
plural gal·li·mau·fries
A jumble; a hodgepodge.

[French galimafrée, from Old French galimafree, sauce, ragout : probably galer, to make merry. See GALLANT + mafrer, to gorge oneself (from Middle Dutch moffelen, to open one's mouth wide, of imitative origin).]
(From The American Heritage® Dictionary of the English Language, Third Edition copyright © 1992 by Houghton Mifflin Company)

What does a Twain control, spiral art program, set of linked list classes, a quick sort routine, a hex editor and a myriad of other custom controls have in
common?

They are all part of Desaware's ActiveX Gallimaufry.

You'll find most of these controls useful, the rest entertaining – but we guarantee that you'll find them all educational, because they come with complete Visual Basic 6.0 source code.

Curious?

Want to learn some advanced API programming techniques? Visit our web site for a full description and demo.

THE CUSTOM CONTROL FACTORY V 4.0
The Custom Control Factory is a powerful tool for creating your own animated buttons, multiple state buttons, toolbars and enhanced button style controls in Visual Basic and other OLE control clients, without programming. With 256 & 24 bit color support, automatic 3D backgrounds, image compression, over 50 sample controls and more. Plus MList2 - an enhanced listbox control. 16 & 32 bit ActiveX controls and 16 bit VBXs included.

INDEX