

SpyWorks™

Version 8

for Visual Basic 6

by

Desaware, Inc.

Information in this document is subject to change without notice and does not represent a commitment on the part of Desaware, Inc. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Desaware, Inc.

Copyright © 1994-2006 by Desaware, Inc. All rights reserved. Printed in the U.S.A.

Desaware, Inc. Software License

Please read this agreement. If you do not agree to the terms of this license, promptly return the product and all accompanying items to the place from which you obtained them.

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

This program will be licensed to you for your use only. If you, personally, have more than one computer, you may install it on all of your computers as long as there is no possibility of it being used concurrently at more than one location by separate individuals. 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, 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. Companies or schools interested in multiple copy licenses or site licenses should contact Desaware, Inc. directly at (408) 404-4760.

Should your intent be to purchase this product for use in developing a compiled Visual Basic program that you will distribute as an executable (.exe) file, review the listing of which files (located below and in the File Description section of the product manual) can be distributed and or modified. If Desaware files are included in your executable program, you must include a valid copyright notice on all copies of the program. This can be either your own copyright notice, or "Copyright © 2005 Desaware, Inc. All rights reserved."

You have a royalty-free right to incorporate any of the sample code provided into your own applications with the stipulation that you agree that Desaware, Inc. has no warranty, obligation or liability, real or implied, for its performance.

SpyWorks Compiled Files: You may include with your program a copy of the files dwsbc80.ocx, dwshk80.ocx, Desaware.shcomponent11.dll, Desaware.shcomponent20.dll, and dwshengine80.dll. You may also distribute DLL files created using the ExportWizard.exe and dwexutil.exe utility programs. You may **not** modify the files listed above in any way.

SpyWorks Source Files: Source code for portions of SpyWorks are included for educational purposes only. You may use this source code in your own applications only if they provide primary and significant functionality beyond that included in the SpyWorks package. You may not use this source code to develop or distribute components and tools that provide functionality similar to all or part of the functionality provided by any of the components or tools included in the SpyWorks package.

Please consult the on-line Help file under the topic File Descriptions for additional information.

Limited Warranty

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

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective CD(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 CD 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, California 95136

Table of Contents

TABLE OF CONTENTS	4
WHAT MAKES A GOOD MANUAL?	8
INTRODUCTION	9
NEW FOR VERSION 8.0.....	9
A NOTE ON FILES	9
INSTALLATION	11
COMPATIBILITY ISSUES	11
SPYWORKS PHILOSOPHY.....	12
KNOWLEDGE, SAFETY AND EASE OF USE.....	13
USING SPYWORKS (PLEASE READ!).....	14
CUSTOMER SUPPORT	15
REGISTER! REGISTER! REGISTER!	15
SPYWORKS CONCEPTS: SUBCLASSING	16
INTRODUCTION TO SUBCLASSING.....	16
HOW MIGHT YOU USE SUBCLASSING?	19
CAUTIONS ON USING SUBCLASSING.....	20
DELAYED EVENTS - POSTING AN EVENT TO YOURSELF	21
16 BIT SUBCLASSING	21
SUBCLASSING AND SPYWARE	21
USING DWSBC80.OCX.....	22
SUBCLASSING MULTIPLE WINDOWS WITH DWSBC80.OCX.....	22
USING DWSBC80.OCX IN A CONTROL ARRAY	23
CROSSTASK ISSUES.....	23
<i>Process Spaces</i>	23
THE DWSUBCLASS OBJECT (DWSHVB8.DLL).....	24
IMPORTANT TIPS	24
<i>Recreating Windows</i>	24
<i>Frozen Messages</i>	25
SUBCLASSING EXAMPLES.....	25
FOR ADDITIONAL INFORMATION ON SUBCLASSING	26
SPYWORKS CONCEPTS: WINDOWS HOOKS	27
TYPES OF HOOKS	29
SHOULD YOU USE HOOKS OR SUBCLASSING?.....	30
KEYBOARD HOOKS AND SPYWARE	31
USING DWSHK80.OCX FOR KEYBOARD HOOKS.....	32
SETTING UP DWSHK80.OCX FOR KEYBOARD HOOKS	32
KEY VALUE FORMAT	33
DISCARDING KEYSTROKES.....	33
USING DWSHK80.OCX FOR WINDOWS HOOKS	33
SETTING UP DWSHK80.OCX FOR WINDOWS HOOKS	33
DWSHK80.OCX - USE OF THE NODEF PARAMETER	34
THE DWGENERICHOOK OBJECT	34
THE DWPRETRANSLATE OBJECT	35
HOOK EXAMPLES	36
FOR FURTHER INFORMATION ON HOOKS.....	39

SPYWORKS CONCEPTS: PRIVATE WINDOWS.....	40
THE DWPRIVATEWINDOW OBJECT.....	40
SPYWORKS CONCEPTS: DWEASY - A MULTIFUNCTION CONTROL.....	41
PLACING THE DWEASY80.OCX CONTROL	41
DWEASY.OCX - MOUSE TRACKING	42
DWEASY80.OCX - DETERMINING UPDATE AREA DURING PAINT.....	42
DWEASY.OCX - TINY CAPTIONS AND ROLLUP WINDOWS	43
<i>Notes on working with Tiny Captions and Rollups.....</i>	<i>43</i>
DWEASY80.OCX - ADDING SCROLLBARS TO FORMS AND CONTROLS.....	44
DWEASY80.OCX - OTHER FUNCTIONS	44
<i>System Menu Support</i>	<i>44</i>
DWEASY80.OCX - WINDOWS EXPLORER/FILE MANAGER DRAG- DROP COMMANDS	45
DWSHELLLINK - SHELL LINK SUPPORT.....	45
<i>Obtaining a dwShellLink Object.....</i>	<i>45</i>
<i>Notes on Working with Shell Links.....</i>	<i>46</i>
DWEASY EXAMPLES:.....	46
SPYWORKS CONCEPTS: DWSHENGINE80.DLL FUNCTION LIBRARY.....	48
DATA AND MEMORY ACCESS	48
USER DEFINED TYPE (UDT) PACKING FUNCTIONS	49
MISCELLANEOUS FUNCTIONS.....	50
DWSPY3X EXAMPLES	50
SPYWORKS CONCEPTS: EXPORTING FUNCTIONS.....	51
WHAT ARE EXPORTED FUNCTIONS?.....	51
HOW DYNAMIC EXPORT TECHNOLOGY WORKS.....	52
THE DWEXPORTER CLASS (FOR VB6)	53
<i>GetFunctionCount</i>	<i>54</i>
<i>GetModuleHandle</i>	<i>54</i>
<i>GetFunctionInfo</i>	<i>54</i>
THE DESAWARE EXPORT UTILITY (FOR VB 6)	55
WHAT ARE RESOURCES? (FOR VB 6)	56
ADDING RESOURCES TO THE ALIAS DLL (FOR VB 6)	57
THE DWExUTIL MAIN FORM (FOR VB 6).....	58
THE EXPORTS CLASS (FOR VISUAL STUDIO .NET).....	59
THE EXPORT WIZARD (FOR VISUAL STUDIO .NET)	59
TESTING EXPORTED FUNCTIONS	60
DISTRIBUTING YOUR COMPONENT.....	60
<i>Distributing the Alias DLL for VB 6.....</i>	<i>60</i>
<i>Distributing the Alias DLL for .NET</i>	<i>61</i>
WARNING! EXPORTING FUNCTIONS IS DANGEROUS!	61
CONTROL PANEL APPLETS AND MULTITHREADING CLIENTS IN VISUAL BASIC 6	61
SPYWORKS CONCEPTS: INTERFACE EXTENSIONS AND HOOKS.....	63
INTRODUCTION TO INTERFACES	63
OVERRIDING INTERFACES	64
REFERENCING NON-AUTOMATION COMPATIBLE INTERFACES	64
HOW TO AVOID CORRUPTING YOUR SYSTEM REGISTRY	65
IMPLEMENTING STANDARD AND NON-AUTOMATION COMPATIBLE INTERFACES.....	65
DECLARING AND INITIALIZING THE DWAXEXTN.DLL COMPONENT.....	66
INSIDE THE SPYWORKS INTERFACE EXTENSIONS	68
THE COM CONTRACT	68
THE TWO SIDES OF COM.....	69

HACKING FURTHER	70
IMPLEMENTING IObjectSafety – THE EASY WAY	71
IMPLEMENTING IObjectSafety – THE EASIER WAY	73
CALLING GENERIC INTERFACES	74
SPYWORKS CONCEPTS: WINSOCK - INTERNET/INTRANET PROGRAMMING	76
HOW TO APPROACH THE WINSOCK PACKAGE	76
IMPORTANT NOTE REGARDING SUPPORT FOR THIS COMPONENT	76
LEARNING WINSOCK	77
<i>IP Addressing</i>	77
<i>Ports</i>	77
<i>UDP and TCP</i>	78
DWSOCK ARCHITECTURE	78
<i>Dependencies:</i>	79
SMTP	80
USING THE WINSOCK PACKAGE	80
WINSOCK UTILITY FUNCTIONS	80
<i>Obtaining Winsock Version Information</i>	80
<i>Obtaining Your Host Name</i>	80
<i>Determine the Standard Port Number of a Service</i>	80
<i>Perform an Asynchronous Name Resolution</i>	80
HTTP EXAMPLE:	81
SPYWORKS CONCEPTS: COMPONENTS AND CLASS LIBRARIES	82
DESAWARE WINDOWS UTILITIES AND SUBCLASSER	82
SPYNOTES	83
SPYWORKS CONCEPTS: BACKGROUND THREADS	84
DWBACKTHREAD - QUICK START	84
DWBACKTHREAD - METHODS AND PROPERTIES	85
<i>LaunchObject(ObjectName As String) As Object</i>	85
<i>BackgroundExecute()</i>	85
<i>BackgroundExecuteDelayed()</i>	86
<i>BackgroundObject</i>	86
DWBACKTHREAD - SUMMARY OF RULES	86
SPYWORKS CONCEPTS: TOOLS AND UTILITIES	88
SPYMSG	88
SPYWIN	88
SPYMEM FOR WINDOWS	89
SPYMENU	89
SPYWORKS CONCEPTS: CALLBACKS	90
DISTRIBUTION AND LICENSING	91
ABOUT OUR SOFTWARE LICENSE	91
LICENSING DWAXEXTN.DLL, DWBKTHRD.DLL, DWSHVB8.DLL, DWSHVB8.DLL, DWSHVB8.DLL, DWSHVB8.DLL, DWSHVB8.DLL, AND DWSOCK6.DLL	91
USING THE LICENSE KEY WITH DWSHVB8.DLL	92
USING THE LICENSE KEY WITH DWSOCK8.DLL	92
USING THE LICENSE KEY WITH DWAXEXTN.DLL	92
USING THE LICENSE KEY WITH DWBKTHRD.DLL	93
CREATING A LICENSE KEY	93
FINAL NOTES ON LICENSING DLLs	93
LICENSING DWSBC80.OCX, DWSHK80.OCX AND DWEASY80.OCX FOR USE AS CONSTITUENT CONTROLS	94

FILE DESCRIPTIONS AND REDISTRIBUTION TERMS	94
SPYWORKS AND VISUAL STUDIO .NET	99
ADDITIONAL TOPICS	100
SHMESSAGES.INI - CONFIGURATION AND INITIALIZATION FILE.....	100
APPLICATION SETUPS	100
MESSAGE INTERPRETATION	100
STYLE INTERPRETATION	101
MESSAGE GROUPING.....	101
TECHNICAL SUPPORT	102

What Makes a Good Manual?

A note from Daniel Appleman, President of Desaware Inc.

When I decided that it was time to revise the SpyWorks printed manual a while back, I faced a serious dilemma. SpyWorks has evolved considerably since its release, growing far beyond the simple subclassing and hook capabilities provided by the earlier versions. Just covering the functionality of the package itself would take a substantial book. Trying to teach the underlying concepts and possible applications of all of the components of this package would take an encyclopedia, for the functionality offered by SpyWorks spans the entire gamut of Windows technology. Indeed, both Desaware's staff and our customers are constantly discovering new applications.

So what really needs to be in the printed documentation?

Let's ignore, for the moment, the increasing trend among software companies to not include printed documentation at all. I personally like printed books, and have not yet become accustomed to reading extensive documentation on-line.

Yet, at the same time, there are certain types of documentation that I am quite satisfied, even prefer to read on-line. Specifically - the details of individual functions; their parameters and operation is often easier to access on-line. This is especially true considering the ability of controls to link directly to help files, and their build-in search and index capabilities.

So I started with the following two premises:

1. That the on-line Help file should be comprehensive and include all of the printed documentation as well.
2. That no matter what I choose to do, I won't please everyone. In fact, I probably won't even please most people.

So I drew inspiration from the old song, if you can't please everyone, you might as well please yourself. Most of the detailed reference material, including lists of properties and events, is only available in the on-line Help file. This is the kind of information that you will generally look up as needed anyway, and I've found that it is much easier to look up this information on-line. The printed manual is designed to place all of the components and features into some sort of context. It contains overview information and feature descriptions to help familiarize you with both SpyWorks and the underlying Windows technology that SpyWorks exposes to Visual Basic programmers. It is intended to be the kind of manual that you can sit down and read and learn something, without distracting you with the nitty gritty details that you are unlikely to memorize in any case.

This is, to the best of my ability, the manual that I would like to read if I purchased a product of this kind. It draws on some of the material from the previous editions, re-organizes it, and adds new material, portions of which were written by me, with additional contributions by Desaware's staff.

I hope you find it satisfactory.

Daniel Appleman

Introduction

SpyWorks is probably the most unusual add-on product available for Visual Basic. As such, it is very important that you review this introduction. It will help you to understand both the features and the limitations of this product.

New for version 8.0

SpyWorks 8.0 is a major product upgrade designed to address three key issues:

- Like any generic subclassing/hook tool, it can be used by unscrupulous developers to create various types of spyware.
- We wanted to substantially improve the handling of hooks under adverse situations – such as dealing with system crashes.
- This release provides support for .NET 2.0.

SpyWorks 8.0 represents a major fork in development of the package. As such, it can be installed on the same system with version 7.1. It contains a completely new set of components and a separate subclassing/hook engine.

Major changes are as follows:

- Anti-spyware technology. These changes are designed to prevent the SpyWorks components from being incorrectly identified as Spyware. This takes two forms: first, the new subclassing and hook controls have built in restrictions that make them unable to intercept keystrokes from certain windows (such as text boxes used to capture passwords) making the components less useful to spyware authors. Second, the components have been renamed to not include the word "spy" or "spyworks" because these were causing confusion among end-users.
- SpyWorks 8.0 has improved handling of Windows hooks, particularly with regard to recovery when applications crash.
- SpyWorks 8.0 does not include a light edition of the NT Service toolkit. This change was made because the full toolkit is included with the Universal COM product.

A Note on Files

The following files may be redistributed. When redistributing these files, they should be installed in the system folder if they were installed in the system folder on your system, otherwise they may be installed in a private folder.

dwshk80.ocx – SpyWorks Windows Hook and KeyBoard Hook ActiveX control. You can use this in Visual Studio .NET projects but we recommend using the Desaware.shcomponent.dll file instead. Refer to the SpyWorksDotNetManual.pdf file's *Migrating to the SpyWorksDotNet component* section for information on migrating to the new component. This file is installed in your System folder.

dwsbc80.ocx – SpyWorks Subclass ActiveX control. You can use this in Visual Studio .NET projects but we recommend using the Desaware.shcomponent???.dll file instead. Refer to the SpyWorksDotNetManual.pdf file's *Migrating to the SpyWorksDotNet component* section for information on migrating to the new component. This file is installed in your System folder.

dwshengine80.dll – SpyWorks Windows Hook and Subclass engine file. Required by Desaware.shcomponent.dll, dwshk80.ocx, dweasy80.ocx and dwsbc80.ocx. This file is installed in your System folder.

dwshutl80.dll – SpyWorks function library, a subset of dwshengine80.dll for those who only require the function library and are not using the controls.

dwshvb8.dll – SpyWorks Windows Hook and Subclass file written in VB6.

dwssock8.dll, dwsmtp8.dll – SpyWorks Winsock and SMTP components written in VB6.

SpyWorks includes sample file projects for Visual Basic 6

Installation

You must have Visual Basic already installed on your system before running this program.

1. Place your SpyWorks CD in your CD ROM drive or find the corresponding directory in your Desaware Universal distribution (or follow the instructions provided in the Universal COM package).
2. The setup program should automatically start if your computer's autorun mode is on. Otherwise, use the Start Button's Run... command in Windows to run d:\setup.exe or e:\setup.exe (depending on the drive letter assigned to your CD ROM drive). You can also use Windows Explorer or File Manager to run this program.
3. The setup program will prompt you for a destination directory for the SpyWorks files. The SpyWorks custom controls and dynamic link libraries will be installed and registered in the appropriate System directory.
4. 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.
5. The directory containing the SpyWorks sample files may contain files readme.txt or readme.rtf. 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, or the Windows WordPad program to view readme.rtf.

Compatibility Issues

SpyWorks extensions use standard Windows techniques for subclassing windows. They do not violate any of the rules or requirements of Windows programming and thus should remain compatible with future versions of 32 bit Windows. SpyWorks 6.0 and later are based upon ActiveX technology and are not compatible with the versions of Visual Basic earlier than Visual Basic 4.0. Some of the new features in SpyWorks 8 are applicable only in Visual Basic 6.0 or later, or Visual Studio .NET or later.

SpyWorks 8 controls have been tested with Visual Basic 6 and Visual Studio .NET 1.1, under Windows 2000, Windows XP, and Vista. The sample code and utilities provided are distributed in Visual Basic 6, Visual Basic .NET (where applicable) and C# (where applicable) formats. Support for Visual Studio .NET 1.0 and VB4 and VB5 along with Windows 98 and ME can be found in SpyWorks 7.1 which is still available.

We obviously cannot guarantee that this product will remain compatible with future versions of Visual Studio, however any changes that would invalidate the use of the SpyWorks controls would likely break any program that uses Windows API functions, and since API functions are used by many Visual Basic programmers and Microsoft's own Visual Studio sample programs, the odds are good that applications that use SpyWorks will continue to work for future versions of Visual Studio.

Please Refer to the On-line Help file for the latest information on migrating SpyWorks controls from Visual Basic 4, 5, 6 or Visual Studio .NET.

Please refer to the on-line Help file for the latest information on compatibility when migrating from SpyWorks 2.1 to the current versions of SpyWorks with Windows NT/2000/XP/Vista, and Windows 95/98/ME.

SpyWorks Philosophy

One of the great advantages of Visual Basic over any other Windows development language is that it is extremely "safe" to use. In theory, it is impossible for a Visual Basic programmer to crash the system or to cause a General Protection Fault, Exception or other system error from within Visual Basic. This makes programming extremely efficient - you can quickly experiment and modify your code, examining intermediate values in the immediate window as needed. This differs markedly from other development systems where bugs frequently cause memory corruption, use of invalid pointers, and other errors that require sophisticated debuggers and many reboot/restart cycles.

Visual Basic protects you from your mistakes, but there is a price to pay for this safety. Visual Basic only implements a subset of the features that are available under Windows. As a result, there are many tasks that are difficult or impossible to do with Visual Basic directly. There are currently two methods for extending Visual Basic. You can directly access Windows API functions from VB, or you can write dynamic link libraries or VB custom controls using C or C++ and traditional Windows programming techniques.

SpyWorks provides a third approach to extending Visual Basic. It consists of a number of tools and programs to support extending the capabilities of Visual Basic from within the VB environment. No C or C++ language or other tools are required. These tools take two forms: Extension tools and Debugging tools.

The major extension tools include the following ActiveX components, classes and dynamic link libraries.

- Dwsbc80.ocx is a powerful custom control that allows you to intercept and manipulate the underlying Windows message stream for any Visual Basic form or control (including most ActiveX controls).
- Dwshk80.ocx contains a task or system keyboard event detector which allows you to intercept keyboard entries before they are received by Visual Basic or another application. Dwshk80.ocx also supports Windows "hooks" and is ideal for intercepting Windows messages on a global basis. Dweasy80.ocx implements common subclassing tasks including Mouse tracking, scrolling and virtual forms, file drag-drop, tiny captions and rollup windows.
- Dwshengine80.dll contains a set of functions that help you take advantage of advanced API techniques and work with the various controls. It is also used by the controls themselves to perform their tasks. SpyWorks 8 includes a .NET class wrapper for some of the functions of dwshengine80.dll.
- Exported Functions - SpyWorks contains Dynamic Export Technology™ which allows you to export functions from your VB created ActiveX DLL's. And it does it without modifying your compiled DLL! Now you can use Visual Basic to create export function libraries, and Control Panel Applet/ExtensionsThis technology has been extended to allow you to export functions from your Visual Basic .NET or C# assemblies
- DwShvb8.dll - SpyWorks includes a VB 6.0 authored DLL for subclassing, hooks, and management of custom windows. It uses the same paranoid approach of our full features controls - to help it coexist safely with other controls that subclass and to provide effective clean-up. It is also designed for high performance with low level message filtering, shared subclass routines, and maximum use of early binding. And it includes complete VB source code.

- **ActiveX Extensions** - ActiveX is about interfaces, and SpyWorks has the ability to let you override the behavior of interfaces that are part of your Visual Basic 6.0 components. For example: you can customize the behavior of the VB property window for controls that you create. You can add new standard interfaces to your components (especially standard interfaces that are not VB compatible), such as IObjectSafety that gives you total control over the safety marking of your ActiveX controls.
- **Winsock Library** - Like most people, we've been fascinated by the Internet. We wondered whether it might be feasible to access the underlying Winsock API directly from Visual Basic without the overhead of ActiveX controls. It is, and this component comes with the source code that demonstrates how it is done. Includes classes for FTP and HTTP support as well.
- **Creating Background Threads** - SpyWorks includes the dwBackThread component which allows you to create objects in independent threads and to make asynchronous method calls on those objects in Visual Basic 6.0.
- **DwTaskbr.ocx** - is a 6.0 authored ActiveX control that is similar to the Windows 95/98/ME or NT/2000/etc. task bar. This control is designed for MDI applications and allows you to switch between MDI childs. This control is also included in Desaware's ActiveX Gallimaufry and includes complete VB source code.
- **DwCmndlg.dll** - is a 6.0 authored ActiveX component that is similar to the Common Dialog ActiveX control. This component exposes the full capability of the Common Dialog library and includes additional functionality beyond the Common Dialog ActiveX control. This component is also included in Desaware's ActiveX Gallimaufry and includes complete VB source code.

SpyWorks includes many more components, utilities and examples - this list just reflects those that we believe are the most important. The remainder of this manual groups the components by features.

There is a price to pay for the power provided by the SpyWorks components. SpyWorks is one of the few products you will ever buy that proudly and clearly claims to make it much easier to crash your program. This tradeoff is common in the programming world: with power and flexibility comes a reduction in the level of protection provided by the environment. SpyWorks, by intercepting the Windows message stream and improving access to the Windows API makes it very easy to crash not only the application that uses it, but other applications as well.

Knowledge, Safety and Ease of Use

As Visual Basic becomes more and more powerful, the degree of knowledge necessary to take full advantage of its capabilities increases dramatically. We feel that a key aspect of the SpyWorks package is educational. Not only should our tools and components allow you to use advanced techniques, they should teach you those techniques as well. Mostly, we do this by providing copious amounts of Visual Basic source code.

But you will also find that there are certain techniques that appear in the general press that we do not use or advocate. This has become more common as Visual Basic has become more sophisticated. The truth is, there are some techniques that are now possible that we believe should not be part of any professional software product. Rewriting virtual table entries on the fly is one of those techniques. Changing the type library declarations of standard system interfaces is another. Even subclassing, a technique that we do support and advocate, is a good example. We believe that in most cases, subclassing is best handled at the component level rather than within an application (even if the component may now be written in Visual Basic instead of C++).

So even as SpyWorks demonstrates some of the most advanced techniques available to VB programmers, the techniques are based on our view of what is appropriate for a stable and supportable application or component. In other words - just because something can be done in Visual Basic doesn't always mean that it should be done in Visual Basic.

Using SpyWorks (Please Read!)

SpyWorks is designed for the intermediate to advanced Visual Basic programmer who has a knowledge of how to use the Windows Application Programmer's Interface (API). Some components of the package are useful to anyone (most of the features in the dweasy80.ocx control are, in fact, very easy to use). However, a good understanding of Windows is required to really use this package successfully.

If you already know Windows well, you will find SpyWorks extremely easy to use. Simply consider the task you wish to perform and how you would do it in C or C++, then write it in Visual Basic. Any code that you would normally write in a Windows procedure in response to a Windows message, you can place in an event in an appropriately configured dwsbc80.ocx ActiveX control. Any time you need to export a function from a VB DLL, look at the dwExport functionality provided by SpyWorks. Any time you need to implement or call an interface that is not directly supported by VB, you can do so using the dwAxExtn.dll component. Where you would use Windows hooks, use dwshk80.ocx.

If you have never programmed in Windows, you must learn about it in order to use this package effectively. The professional version of Visual Basic comes with the help file for the Windows API - this can be used as a reference for all available Windows API functions and Windows messages.

We also recommend *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"* as a supplementary text for working with SpyWorks. This book was written by the author of SpyWorks, and in fact includes a demonstration edition of the SpyWorks controls. You may also find the book "Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed" to be helpful.

SpyWorks is a tool. Most add-on programs have a clearly defined set of operations that they can perform. Their documentation can, and often does, include extensive examples to show the capabilities of the product. A dozen books and manuals could not begin to do this with SpyWorks, because it has no clearly defined set of operations. It is a can-opener that enables you to tap the full power of Windows from within Visual Basic. This manual includes a number of examples of how the extension controls can be used, but we cannot even begin to guess at the potential of what can be accomplished.

Customer Support

SpyWorks requires an understanding of the Windows API. We at Desaware will gladly and enthusiastically fix any bugs in our software that pass through our screening process. However, due to the nature of this product, we cannot possibly resolve all issues that relate to use of the Windows API and possible incompatibilities between the Windows API functions and Visual Basic.

What we can do is this: If you want to do something and think you have an approach, or have a problem and would like some direction, feel free to drop us a line by fax or email (contact information is located in the Register! and Technical Support sections of this manual). If it appears to be a bug in our software, we will drop everything to fix it and send you updated software. Otherwise, if it is something we can answer quickly, we'll fax or email an answer to you as quickly as possible. If it is something that is a more extensive problem, we may propose to solve it on a consulting basis.

If you have purchased this software directly from Desaware and have read this introduction and you feel that SpyWorks is not for you, 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). 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 (or send you upgrades if you have the Subscription Edition). It will also allow us to send you information about SpyWorks add-ons and other Desaware products.

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

Desaware, Inc.
3510 Charater Park Dr. Suite 48
San Jose, CA 95136
USA
Phone: 408/404-4760, Fax: 408/404-4780
Email: support@desaware.com

We also invite you to subscribe to our email listserver by sending a message to listserve@desaware.com and including the word "Subscribe" in the subject line. We promise we won't send you email unless we have something really important to share.

SpyWorks Concepts: Subclassing

Subclassing refers to the process of intercepting Windows messages that are normally processed behind the scenes. More information on this technology will be discussed shortly.

There are a number of approaches to subclassing. As of Visual Basic 5.0, it became possible to do subclassing within an application using Visual Basic alone. Unfortunately, the limited documentation included with Visual Basic demonstrated perhaps the worst possible way to subclass a window. It did not discuss limitations relating to subclassing of windows in other processes. It did not address the dangers of subclassing, and the fact that simple subclassing techniques can not only interfere with the normal Visual Basic debugging process, but the Visual Basic environment itself. It did not discuss how subclassing should be done in order to minimize interference with other controls that use subclassing. It did not discuss how to relate windows with objects.

To handle the entire spectrum of subclassing requirements, SpyWorks includes two completely different approaches to subclassing. The `dwsbc80.ocx` control is a high end control written in C++ that supports advanced subclassing features including subclassing of windows across applications (cross-process subclassing). The `dwshvb8.dll` component is an ActiveX automation component (not a control) which is in many ways more efficient than a control and is written almost entirely in Visual Basic. This component includes source code. If you prefer not to use a subclassing component, you can perform your design and testing using the component, then add the necessary classes directly to your application for final release.

These two components will be discussed shortly in detail. First, let's take a look at the process of subclassing itself.

Introduction to Subclassing

Under Microsoft Windows, every window has a special function called a window function. This function has four parameters as follows:

Long WndProc(window handle, message number, wParam parameter, lParam parameter)

In a 16 bit environment, the window handle, message number and wParam parameter are 16 bit integer values. All parameters are 32 bit under Win32. The term "sending a message" to a window means that the window function has been called for that window. Each possible message has a message number, and a message can have up to two parameters. The lParam parameter is frequently used to pass a pointer to a larger data structure, so it is possible to include a great deal of information in a message.

Windows defines many standard message numbers. Message numbers above &H400 are called user-defined, which means that they depend on the type of window. It is also possible to define a type of message called a "registered" message. A registered message is identified by a name (or text string). Windows allocates a registered message number for each unique registered message.

Messages are called from several sources. The windows environment sends messages indicating that system events have occurred. For example: when a window needs to receive information on mouse movement or if a key has been pressed when a window has the focus, Windows will send the appropriate mouse or keyboard messages to the window. Windows also sends messages to a window to instruct it to perform certain tasks such as erasing its background or painting its client area.

Windows programmers frequently send messages to windows to instruct them to perform tasks as well. For example: adding and deleting text in an edit control or list box is accomplished by sending messages to the control. You can send messages to windows using the Windows API `SendMessage` and `PostMessage` functions.

When you use the `SendMessage` function to send a message, the windows function for the window is called immediately. The result of the `SendMessage` function is set to the value returned by the windows function. When you post a message to a window, the message is added to a system message queue. The windows function for the window will be called in due course when the message is processed by Windows. Obviously in this case it is impossible for the windows function to return a value, since the application that posted the message has long since past the point where it posted the message. In other words, when you use `SendMessage`, your program does not continue to run until the message has been completely processed. When you use `PostMessage`, your program continues to run immediately - the message will be processed later.

There are dozens (if not hundreds) of possible messages. It would be terrible if each window function had to implement all of the code necessary to process each message. Fortunately, Windows provides default processing for most messages. Each window function processes only those messages that it needs to.

Under traditional Windows development, you can subclass any window by forcing Windows to call a function you define before it calls the true window function for that window. You then have the opportunity in your function to process any messages yourself. You can then either return directly to Windows, or allow the original window function to execute.

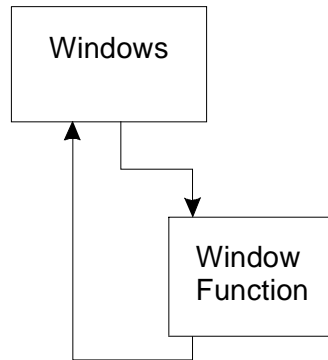


Figure 1
Windows Functions

The SpyWorks subclassing components support several types of subclassing. The most common involves detection before default processing occurs (pre-default processing). This means that the component gets message information before the window function for the form or control does. This is shown in Figure 2.

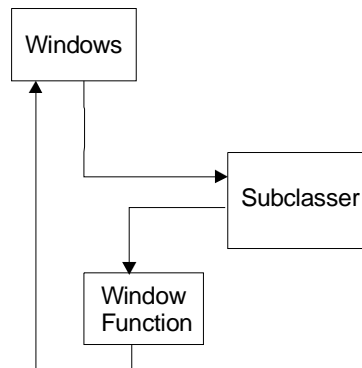


Figure 2
Subclassing before default processing

As you can see, the component gives you the option as to whether or not the original (default) windows function should be called. In other words, you can, if you wish, completely replace the default processing for any windows message for any window, form or control.

This technique is especially powerful when you consider that you can subclass windows in other applications than your own (dwsbc80.ocx only).

The SpyWorks subclassing components also allow you to specify that messages should be intercepted after the default windows function has been called (post-default processing) as shown in Figure 3.

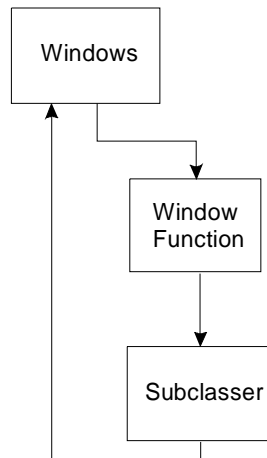


Figure 3
Subclassing after default processing

You can also indicate that specific messages should be posted to the dwsbc80.ocx control (posted message processing). This is common when you need notification that a message was received, but do not need to perform any processing immediately. This is the safest type of subclassing

This technique is especially powerful when you consider that you can subclass windows in other applications than your own (dwsbc80.ocx only)..

How Might You Use Subclassing?

There are a number of common approaches to using subclassing.

Detecting Events:

In this case "events" refer to things that occur in the system that Visual Basic does not allow you to detect directly. For example: your application's main window will receive a message whenever certain system setting changes occur. You can use subclassing to detect these changes. For example: The WM_SETTINGCHANGE message indicates that a system parameter was changed by some application using the SystemParametersInfo API function. In most cases, you will use Posted detecting for this type of subclassing, because you're only interested in detecting when the message arrives and have no need to block or interfere with the normal processing of that message.

Another detection example is when you use API commands to add entries to your application's system menu and wish to detect when the user selects your new menu commands. Visual Basic does not itself allow you to intercept the WM_SYSCOMMAND message. You might also use this to detect when menu commands are invoked in other applications, but you'll need to use the dwsbc80.ocx control in this case, as dwshvb8.dll will only subclass windows within your own application.

Overriding Message Behavior:

You may want to actually change the response of a window to a certain message. This is an extremely powerful technique, as almost all of the behavior of a window is determined by its response to windows messages. If you intercept a message, you can write in your own behavior for the message and actually prevent the message from being forwarded to the window. An example of this is when you wish to create your own context menu for a control (the popup menu that appears when you right click on the control). You can intercept the WM_CONTEXTMENU message before it arrives at the window, using pre-default subclassing. If you block the message, the existing context menu will be disabled. You can bring up your own popup menu during the message processing to effectively create your own context menus.

You can also turn standard controls in some cases to owner draw controls, where you keep the full capability of the standard control while completely overriding the appearance of the control.

Monitoring Messages and their Results:

Sometimes you will want to intercept a message, allow default processing to occur, but check the result returned by the default message processing before allowing the message function to return. An example of this might be intercepting the WM_NCHITTEST. The default message processing returns a code that indicates what type of window element the mouse is over. For example: is it over the caption, client area, minimize box, etc. By using Post-default detection, you can look at the result of this message, then override the return value, tricking a window into thinking the mouse is over the window caption even though it is actually over the client area. This provides a quick way to allow you to reposition a window by dragging the client (instead of the caption).

As you will note, choosing the type of subclassing is a critical decision. You should always use Posted detection if possible. But since it does not allow you to return values or modify the message or its parameters, there are many cases where you will need to use pre-default or post-default processing.

The type of subclassing can be set using the **Type** property on the dwshbc80.ocx control or the **SubclassType** property of the dwSubClass object.

Remember that you can subclass a window multiple times using the SpyWorks components. It is not uncommon to use all three types of message detection on the same window simultaneously to accomplish different tasks. For efficiency sake, the SpyWorks subclassers will actually subclass the window only once in these cases, automatically dispatching events to the appropriate control or object events as needed.

Cautions on Using Subclassing

When you are subclassing a message, and you are using pre-default or post-default processing (not posted), the component raises an event immediately - while the underlying windows message is being processed. The underlying Windows operating system may be expecting the application to both take and avoid certain actions during message processing, depending on the message. Code that you execute at this time poses the greatest risk to your application and the system. For this reason, you should attempt to minimize the code that runs during the event. Also avoid complex tasks such as loading or unloading forms or controls, launching other applications, and so on.

NEVER use the DoEvents function during a non-posted message. Also, you should never use a Message Box during a non-posted message.

Use `Debug.Print` to obtain a debug trace instead of using message boxes or setting a break point. These limitations do not apply when events are triggered by posted messages.

Specific messages may have additional restrictions. Refer to your Windows API reference for further information.

Delayed Events - Posting an Event to Yourself

Sometimes you'll find that there is a need to post a message to your own application. For example: you may have broken up a long operation into small pieces and you want to trigger an event that will occur during normal Windows processing without setting a timer control. Another example is when you are subclassing a window using pre-default or post-default message processing, and wish to perform an operation (such as closing the application) that is not safe during the subclassed event itself.

In both these cases, it would be nice if you can post an event to yourself - simply specify that an event will occur later during normal Windows processing. We call this posting a delayed event.

The SpyWorks components make it easy to accomplish this task. The `dwsbc80.ocx` control uses the **PostEvent** property to accomplish this. Simply assign the property a value, and a **DelayedEvent** event will be raised as soon as the message is processed by the system. The `dwshvb8.dll` component has a similar **PostEvent** and **DelayedEvent** property as part of its `dwPrivateWindow` class.

16 Bit Subclassing

SpyWorks 8.0 no longer includes 16 bit subclassing controls. However, they are still available in SpyWorks 2.1 which is one of the archived products included with the Desaware Universal COM package.

Subclassing and spyware

One of the problems that has occurred in the past with regards to the kind of cross-process subclassing supported by SpyWorks is that while it has numerous legitimate uses, it can also be used by spyware to capture information that end users might wish to keep private (account passwords, for example). Unfortunately, some spyware vendors have used our components in the past in this manner, and as a result some anti-spyware programs have incorrectly blamed our components rather than the client application and added our components to their spyware lists.

SpyWorks 8.0 places some functional limitations in the package that should have no impact on legitimate users, but make the components useless to spyware developers.

With regards to subclassing, the subclassing engine checks all intercepted keyboard and character (`WM_CHAR`) messages to see if the message is destined to the client application (the one that placed the subclass). If so, it is always allowed through. Thus there are no limitations to subclassing your own application.

If the message is from another process, a filter is applied:

- If the destination of the message is a text box with the password style set, the message is not forwarded to the subclasser.
- If the destination is a browser window, the message is not forwarded to the subclasser. The engine applies this filter to the Internet Explorer 6.x, Netscape, Mozilla, Opera and Firefox browsers.

Non character keystroke messages are generally allowed, as are control and alt character combinations.

Using DWSBC80.OCX

Subclassing using the dwsbc80.ocx control is a very simple process.

1. Select the messages to detect.

Dwsbc80.ocx only detects messages that you specify. This helps keep the overhead in subclassing to an absolute minimum. Use the **Messages** and **RegMessage** properties to specify the messages to detect. The **RegMessage** properties allow you to specify a registered message by the name of the message instead of the number. These properties can be used at design time or at runtime. If you do not specify any messages, the control will detect all messages going to the subclassed window.

2. Choose the window, control or form to subclass.

You can use the **CtlParam** or **HwndParam** properties to specify which window, form or control to subclass. You can also add windows or controls to a built-in subclassing array which allows a single dwsbc80.ocx control to subclass many windows or controls.

3. Choose the type of subclassing.

The **Type** property is used to specify whether you want messages detected before the default window function, after the default window function, or simply posted to the dwsbc80.ocx control.

Once you have performed these steps, the dwsbc80.ocx control will receive messages based on the property settings. Messages sent from Windows will trigger the **WndMessage** event or the **WndMessageX** event.

For example: the **WndMessage** event looks like this under the 32 bit dwsbc80.ocx control in VB 6:

```
WndMessage(hwnd As Long, msg As Long, wp As Long, lp As Long, retval  
As Long, nodef As Integer)
```

The window handle is in the hwnd parameter. The message number can be found in the msg parameter. wp and lp are the standard windows wParam and lParam parameters, and their values depend on the individual message. If you are doing a posted message detection, these are the only parameters that you will use.

If you are using pre-default subclassing, you can actually change the values of these parameters and change the message before it is sent to the default message function. If you set the nodef parameter to True, you can block the default message processing from taking place and specify your own return value by setting the retval parameter.

If you are using post-default processing, the retval parameter will already be set to the return value provided by the default window message processing.

Subclassing Multiple Windows with DWSBC80.OCX

The dwsbc80.ocx control has the ability to subclass multiple windows or controls with a single dwsbc80.ocx control. In order to ensure compatibility with the previous versions of the control, this capability was added by incorporating a subclassing array into the dwsbc80.ocx controls. This is an array that can be loaded at runtime. It works completely independently from the standard **HwndParam** and **CtlParam** properties. You can use either or both techniques for specifying windows to subclass. The biggest advantage to the **CtlParam** property is that it is possible to set the property at design time.

The **AddHwnd** property is used at runtime to add windows to the subclassing array. The **RemoveHwnd** property can be used to remove windows from the subclassing array. The **HwndArray** and **HookCount** properties can be used to determine which windows are currently being subclassed.

It is important to recognize that the non-subclassing array properties and the subclassing arrays implement two completely different subclassing subsystems. It is very possible for the same window to be specified using both techniques and thus to be subclassed twice (in which case each message will be triggered twice).

The other dwsbc80.ocx properties that specify messages, detection type, etc. apply to all windows or controls being subclassed by the control.

Using DWSBC80.OCX in a Control Array

Generally speaking, when you create a new control in a control array, that control inherits all of the properties from the original control. While you can create control arrays with the dwsbc80.ocx control, you will need to select the window or control to subclass for each new dwsbc80.ocx control. The list of messages to detect is not copied to the new control. You can easily copy the list of messages from one dwsbc80.ocx control to another using the **MessageArray**, **MessageCount** and **Messages** properties. Registered messages are copied with the control.

Please refer to the on-line Help file for more detailed information on these properties.

CrossTask Issues

Subclassing other applications under Win32 is much more complex than it was under Win16. This is largely due to the fact that Win32 operating systems are multitasking. When you subclass a different application, every time a message arrives in the application that you want to see, the other application must be suspended and control passed to your application. (Note: message filtering takes place in the context of the subclassed application, which limits time consuming task switches only to those messages for which you specifically request.)

What happens if a message is detected in another application but your application is tied up on a long operation such as a long loop? Under Win16, this would never be a problem, as another application could only run when your application was not within an event or was in a DoEvents function. But under Win32, the other application becomes suspended and must wait until your application is ready to process the message. If your application is blocked or even crashed, the other application might become permanently blocked. This can be even more serious with the dwshk.ocx control, where all of the messages in the system can become blocked while waiting for a single application.

For this reason, both the dwsbc80.ocx and dwshk.ocx controls include a **CrossTaskTimeout** property which allows you to limit the amount of time that the other application will wait until your application is ready to process the message.

Process Spaces

Under Win32, each process has its own memory space. Let's say you intercept a message going to another window which has as one of its parameters a memory address. This memory address will be meaningless to your application. In order to facilitate data transfer between processes, SpyWorks includes a number of cross-task memory function in the dwshengine80.dll library (more on this later). However, the dwsbc80.ocx control also includes the **GetAnsiString** and **GetUnicodeString** methods to allow you to easily retrieve text information from other process spaces.

The dwSubClass Object (dwshvb8.dll)

The dwSubClass object in the dwshvb8.dll component is used to perform subclassing of any window in your application (it does not support cross-task subclassing). Use of this component requires Visual Basic version 6.0. The steps for using this object are as follows:

1. Add a reference to the "Desaware SpyWorks 8.0 VB Subclasser" object dwshvb8.dll using the Project-References menu.
2. Dimension a dwSubClass object with events as follows:

Dim WithEvents myobject As dwSubClass

3. During form load or before using the object, create the object as follows:

Set myobject = New dwSubClass

4. Select the messages that you wish to intercept using the **AddMessage** property as follows:

myobject.AddMessage messagenumber

Where messagenumber is a message number to intercept. You may add as many messages as you wish. NOTE: If no messages are specified, the dwSubClass object will intercept all messages.

5. Select the window that you want to subclass as follows:

myobject.HwndParam = windowhandle

6. Add code to the **myobject_WndMessage** event **myobject_WndMessage** for the messages that you are intercepting.
7. Delete the object during form unload or when finished using the object as follows:

Set myobject = Nothing

Important Tips

Recreating Windows

Subclassing for a form or control stops automatically when the control or form is unloaded (destroyed). It is theoretically possible for a custom control to destroy and then recreate its window during normal operation (this is done in some cases when there is the need to change the style for a control). This will cause the subclassing link to that control to end.

One way to handle this is to determine what condition causes this to occur and simply reinstall the subclassing hook to the control.

Another way to accomplish this is to detect the WM_DESTROY message to determine when the window is being destroyed, and post a message to yourself using the **PostEvent** property so that you can later recreate the hook.

A third way to accomplish this with the dwsbc80.ocx control is to set the **Persist** property to True; the subclass control will then automatically attempt to recreate the hook if a subclassed window is destroyed and recreated.

Frozen Messages

There are times when Visual Basic is not able to process messages. The dwsbc80.ocx control can detect this condition and queue messages for later posting. Set the **PostOnFreeze** property to True to enable this functionality, and the **PostOnFreezeMax** property to specify the maximum size of the message queue.

Subclassing Examples

- This example demonstrates how to detect when a ComboBox has been closed (i.e. Drop Up). The ComboBox is named Combo1 and is located on a form named Form1.

1. Add the Desaware dwsbc80 v8 Subclassing Control component to your project. Add a Subclass control to Form1.
2. Select the SubClass control's **Messages** property to display the Select Messages form and Add the WM_COMMAND message to the Selected Messages list. We highly recommend *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API "* as a reference guide to the Windows messages.
3. In **Form_Load** event of Form1, assign the **HwndParam** property to Form1's window handle as follows:

```
SubClass1.HwndParam = Form1.hWnd
```

4. Attach the following VB code to the SubClass control's **WndMessage** event.

```
Dim item As Integer, notification As Integer

' If you are detecting more than one Windows
' message, you want to compare the msg parameter
' here to determine which message you received.
' Since we are only detecting a single message, we
' can skip that step.

' Split the long into two integers.
dwDWORDto2Integers wp, item, notification

Select Case notification
    Case CBN_CLOSEUP: ' CBN_CLOSEUP is declared as Public
        Const CBN_CLOSEUP& = 8
        ' The Combo box was closed, if you have
        ' more than one combo box on the Form,
        ' you will need to compare the lp parameter
        ' to each combo box 's hWnd property to
        ' determine which one was closed.
        Debug.Print "Combo Box Closed"
End Select
```

- This example demonstrates how to detect when a particular window in another application has been closed. This example assumes that the window already exists.
1. Add the Desaware dwsbc80 v8 Subclassing Control component to your project. Add a Subclass control to Form1.

2. Select the SubClass control's **Messages** property to display the Select Messages form and Add the WM_DESTROY message to the Selected Messages list. We highly recommend *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"* book as a reference guide to the Windows messages.
3. In **Form_Load** event of Form1 (or where appropriate), retrieve the window handle (there are several methods you can use to retrieve a window handle based on several criteria – one method is to use the FindWindow API function, another is to use the EnumWindows API function) of the window – for testing purposes you can create a new Visual Basic application where the main form sets its caption to the **hWnd** property value during the **Form_Load** event. Assign the SubClass control's **HwndParam** property to the window handle as follows:

```
SubClass1.HwndParam = hWnd_of_another_application
```

4. Attach the following VB code to the SubClass control's **WndMessageX** event.
Debug.Print "Window was destroyed"

For Additional Information on Subclassing

Additional information on subclassing can be found in the book *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"*.

Many SpyWorks examples demonstrate subclassing techniques. Perhaps the best demonstrations can be found in the spydem32.vbp, and spy8demo.vbp projects.

Refer to the on-line reference for a complete list of properties and events for the dwsbc80.ocx controls and the dwSubClass object of the dwshvb8.dll component. Also, refer to the online reference "Subclass" keyword for additional samples demonstrating subclassing.

SpyWorks Concepts: Windows Hooks

Subclassing is based on the idea of intercepting message by changing the function that is associated with a window, forcing messages going to a window to run your code instead of the function originally assigned to a window. You then have the option of calling the original window function if you wish.

Subclassing suffers from two primary limitations:

1. You must explicitly subclass each window for which you want to receive messages.
2. Subclassing always intercepts messages right before the window function is about to be run.

Hence, Windows provides another mechanism for intercepting messages called Windows hooks. There are a number of different types of Windows hooks available. To see how they work, consider for a moment how messages are generated. This is illustrated in Figure 4. Because messages are generated in many different ways, let's start from the end when a message arrives at a window.

A message arrives at a window when the windows "window function" is called. Subclassing is the process of replacing one window function with another.

There are two ways for a window function to be called. One is through the SendMessage API function. This function causes a window function to be called immediately. The SendMessage API function does not generally return until the window function has completed its operation, and the SendMessage API returns the same value returned by the window function.

A window function is also called by the system when an application calls the GetMessage API function. This is done behind the scenes in Visual Basic, so most VB programmers are not aware that every application is constantly running an infinite loop called a "dispatch loop" which does not exit until the application terminates. This loop calls the GetMessage API to see if any messages are available in the application's message queue. If a message is available, the system determines the destination window and calls the window function with the message.

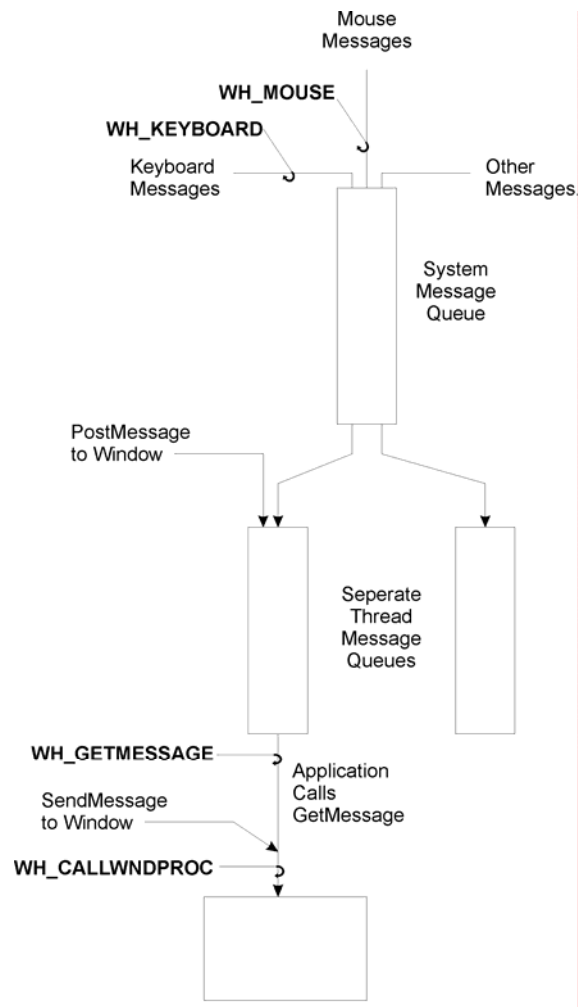


Figure 4
Message Flow and Hooks

The SendMessage and GetMessage message paths are both shown at the bottom of Figure 4. The figure illustrates two of the most commonly used hooks. The WH_GETMESSAGE hook traps messages whenever an application calls the GetMessage API function. This provides a way for you to examine messages that have been posted to an application's message queue before they are processed by the application.

The WH_CALLWNDPROC hook traps every message that goes to a window function, regardless of whether it comes in due to a call to the GetMessage API or a SendMessage call.

But why would you want to use a hook instead of subclassing? Is being able to tell the difference between sent messages and dispatched messages a big enough difference?

Certainly not - you will rarely care where a message comes from.

No, the trick is this: both the WH_CALLWNDPROC and WH_GETMESSAGE hooks allow you to intercept messages going to every window for a particular thread with one operation. In fact, they can allow you to intercept messages going to *every window in the system* just as easily.

This is part of the power of hooks - their ability to tap into the flow of messages before they are dispatched to individual windows.

Continuing with Figure 4, as you proceed up the page, you'll see that each system thread has its own message queue which is fed from a system queue. The system queue receives messages from a number of different sources. The most common of these are keyboard messages, mouse messages and miscellaneous system messages.

The WH_KEYBOARD and WH_MOUSE hooks allow you to trap keystrokes and mouse events before they are actually placed into the system queue. Here too, you have the ability to trap these messages on a thread or system basis with one operation.

SpyWorks provides two components for implementing system hooks. The dwshk.ocx control is available in both 16 bit and 32 bit OCX editions. The dwGenericHook and dwPretranslate objects of the dwshvb8.dll component also provide Windows hook capabilities, though they are not as advanced or flexible as those supported by the dwshk control. SpyWorks 2.1 divides hooks into two controls, sbchhook.vbx and sbckbd.vbx, the latter of which is designed to support keyboard hooks only.

Types of Hooks

The SpyWorks Windows hooks controls support most current types of Windows hooks. Of these, the most likely ones that you will use are the WH_GETMESSAGE, WH_MOUSE, WH_KEYBOARD and WH_CALLWNDPROC hooks. Refer to the on-line component reference and the **HookType** property of the dwshk control for details on how to use these hook types. Note that the dwshk control raises different events for different types of hooks. This is also covered in the on-line documentation for the **HookType** property.

Setting	Description
0 - WH_GETMESSAGE	Implements a WH_GETMESSAGE hook. This hook is triggered any time a Windows function called GetMessage is called during the main message handling loop of a Windows application. It does not detect every message received by a window function, but it is very efficient.
1 - WH_MOUSE	Implements a WH_MOUSE hook. This hook is triggered by mouse events.
2 - WH_MESSAGEFILTER	Implements a WH_MSGFILTER hook. This hook is triggered any time a non-system message is sent to a dialog box, message box or menu.
3 - WH_SYSMESSAGEFILTER	Implements a WH_SYSMSGFILTER hook. This hook is triggered any time a system message is sent to a dialog box, message box or menu.

4 - WH_CALLWNDPROC	Implements a WH_CALLWNDPROC hook. This hook is triggered any time a message is sent to a window function. This hook type does detect every windows message. Even with the advanced filtering used by SpyWorks, use of this hook can impact system performance and should be avoided if possible.
5 - WH_CBT	Implements a WH_CBT hook. This hook is used to implement computer based training applications, providing information on a variety of windows events.
6 - WH_JOURNALPLAYBACK	Implements a WH_JOURNALPLAYBACK hook. This hook is used to simulate keyboard and mouse events to the system, typically after being recorded using the JournalRecord hook.
7 - WH_JOURNALRECORD	Implements a WH_JOURNALRECORD hook. This hook is used to record keyboard and mouse events on the system, typically to implement a macro recorder.
8 - WH_SHELL	Implements a WH_SHELL hook. This hook is triggered when the shell application is about to be activated and when a top-level window is created or destroyed.
9 - WH_CALLWNDPROCRET	Implements a WH_CALLWNDPROCRET hook. This hook is triggered any time a window function returns from a message. This hook type does detect every windows message. Even with the advanced filtering used by SpyWorks, use of this hook can impact system performance and should be avoided if possible. This hook is not supported under NT 3.51.
10 - WH_MOUSE_LL	Implements a WH_MOUSE_LL hook. This hook is triggered by mouse events.
11 - WH_FOREGROUNDIDLE	Implements a WH_FOREGROUNDIDLE hook. This hook is used to detect when the foreground thread is about to become idle.

Should You Use Hooks or Subclassing?

We are often asked whether it is more appropriate to use hooks or subclassing in a given application. While it is not possible for us to make specific recommendations that are right for every application, here are a few general rules that should prove helpful.

You are only interested in messages going to one or two windows.

In most cases you will use subclassing in this situation. The one exception is when messages are being blocked by other parts of the system before they get to the window. For example: if you are creating an ActiveX control and you want it to respond to arrow keys, you will find that the arrow keystrokes are blocked by the container. The most common way to intercept those keystrokes is by using a `WH_GETMESSAGE` hook. This particular situation is handled nicely by the `dwPretranslate` object in the `dwshvb8.dll` component which is designed expressly for this purpose.

You are interested in monitoring messages to a large group of windows, such as all of the controls on a form.

A hook may be most useful in this case, as it eliminates the need to enumerate and subclass individual windows.

You are interested in responding to particular messages regardless of which application is currently active.

A common application for this is implementation of system hotkeys, or monitoring which application has the focus. In this case a `Windows` hook is usually the best solution. Try to avoid using the `WH_CALLWNDPROC` hook, however. It is the most invasive of the hooks and can impact system performance and stability (especially if you have any bugs in your hook code).

In general:

- Try to use subclassing before hooks.
- Try to use thread specific hooks before application wide hooks.
- Try to use application wide hooks before system hooks.
- Use `WH_GETMESSAGE` hooks before `WH_KEYBOARD` hooks and `WH_MOUSE` hooks.
- Use any type of hook before `WH_CALLWNDPROC` hooks.

Keyboard hooks and spyware

One of the problems that has occurred in the past with regards to keyboard hooks is that while they have numerous legitimate uses, they can also be used by spyware to capture information that end users might wish to keep private (account passwords, for example). Unfortunately, some spyware vendors have used our components in the past in this manner, and as a result some anti-spyware programs have incorrectly blamed our components rather than the client application and added our components to their spyware lists.

SpyWorks 8.0 places some functional limitations in the package that should have no impact on legitimate users, but make the components useless to spyware developers.

With regards to hooks, the hook engine checks all keyboard and message hooks to see if the detected event is a keystroke or character message. If the keystroke or character is destined to the client application (the one that placed the hook) it is always allowed through. Thus there are no limitations to hooking or subclassing your own application.

If the keystroke or message is from another process, a filter is applied:

- If the destination is a text box with the password style set, the keystroke or message is not forwarded to the hook.
- If the destination is a browser window, the keystroke or message is not forwarded to the hook. The engine applies this filter to the Internet Explorer 6.x, Netscape, Mozilla, Opera and Firefox browsers.

Non character keystrokes are generally allowed, as are control and alt character combinations.

Using DWSHK80.OCX for Keyboard Hooks

The dwshk.ocx control contains two separate subsystems, one for keyboard hooks, the other subsystem for all other types of hooks. We recommend that you only enable one of these subsystem per control.

Dwshk80.ocx is designed to hook into the Windows keyboard processing system in order to detect keyboard events before they are processed by an application.

Several types of keyboard hooks may be placed, depending on the setting of the **KeyboardHook** property. One type intercepts only keystrokes sent to the process which contains this custom control. Another type intercepts all keystrokes in the system. A third type intercepts keystrokes from a specified process, while a fourth type intercepts keystrokes from a specified thread. Keyboard hooking can also be disabled by the proper setting of the **KeyboardHook** property.

Keystrokes may be processed immediately by the application, or posted for later use. The **Keys** property can be used to set up a filter for keystroke processing. Only keys that are specified will be detected. This significantly reduces the overhead in situations where you are searching only for a few specific key combinations.

Dwshk80.ocx uses a Windows keyboard hook to detect keyboard events. As such, it detects the keys before they are seen by Visual Basic or any other application. This means that you can detect unusual key combinations such as enter, tab and control-break as well as other characters. It also means that if you are not careful, it is possible to completely lock out the keyboard.

The Dwshk80.ocx control can intercept both regular and low level keyboard hooks.

Of course, if you wish to lock out the keyboard, go right ahead and do so.

Setting up DWSHK80.OCX for Keyboard Hooks

Receiving keyboard events using the dwshk80.ocx control is a very simple process.

1. Choose the scope of the hook.

The **KeyboardHook** property specifies the scope of the keyboard hook.

2. Choose the type of notification.

The **KeyboardNotify** property determines when the **KbdHook**, **KeyDownHook**, and **KeyUpHook** events will be triggered for a keyboard event. You can trigger key events when the keyboard activity takes place or have it posted for later use.

3. Select the keys to intercept.

The default is for dwshk.ocx to detect all keystrokes. However, if you are searching only for a specific set of key combinations, you can use the **Keys** property to select the keystrokes to intercept. Using a keys filter in this manner will improve performance.

4. Add event code.

Add your code to the **KeyDownHook**, **KeyUpHook**, or **KbdHook** events. Use the **KeyboardEvent** property to determine whether you will use the **KeyDownHook** and **KeyUpHook** combination, or the **KbdHook** event.

Key Value Format

A key value in dwshk80.ocx is represented by a long value. The low 16 bits corresponds to the virtual key number (which is identical to the **KeyCode** parameter in the VB **KeyDown** and **KeyUp** events).

The high 16 bits determine the requested state of the SHIFT, CONTROL and ALT keys where bit 0 corresponds to the state of the SHIFT key, bit 1 corresponds to the state of the CTRL key and bit 2 corresponds to the state of the ALT key. This matches the **shift** parameter to the VB **KeyDown** and **KeyUp** events.

Discarding Keystrokes

The keyboard events have two parameters that are significant when it comes to discarding keystrokes. Setting the **Discard** property prevents subsequent hooks from seeing the keystroke. But you should also set the keycode parameter to zero to make sure that the original keycode is not forwarded to the application.

Naturally, the **KeyboardNotify** property must be set to detect the keystrokes when hooked. You cannot remove keystrokes if the notification is posted.

Using DWSHK80.OCX for Windows Hooks

The dwshk80.ocx control includes a separate subsystem for handling non-keyboard hooks. The dwshk80.ocx control provides limited ability to modify or to discard messages. The limitations depend on the types of hook, not the control itself. Unlike subclassing with the dwsbc80.ocx control, you cannot return a result to Windows.

Because Windows hooks do not require a window handle, it is possible for the dwshk80.ocx control to detect the WM_NCCREATE and WM_CREATE messages that occur when a window is created. This makes it possible for the first time to change the style of a newly loaded VB form or control during the creation process. Note that during the WM_CREATE or WM_NCCREATE message, Visual Basic events are frozen for the application. This means that you can only detect these messages for your own application as posted (or you can select the **Notify's** property to "When Hooked" and set the **PostOnFreeze** property to True).

Because of the technology involved in Windows hooks, dwshk80.ocx is not able to detect messages going to VB Graphical controls or to detect internal Visual Basic messages.

By the way, the dwshk80.ocx control includes the same **Delayed** event mechanism implemented by the dwsbc80.ocx control allowing you to efficiently post messages to yourself.

Setting up DWSHK80.OCX for Windows Hooks

Receiving windows hooks using dwshk80.ocx is a very simple process.

1. Select the type of hook.

Use the Hook property to select the type of hook you wish to use.

2. Select the messages to hook.

Use the Messages property to bring up the messages dialog box to select messages to intercept. You can also use the MessageArray, Messages, and MessageCount properties to dynamically set messages to detect at runtime. These properties work identically to the properties of the same name in the dwsbc80.ocx control. If you do not specify any messages, the control will detect all messages.

3. Set the scope of the hook.

Use the Monitor property to specify the scope of the hook. You can hook a single thread, any one form (with or without its child windows), a single process, or the entire system.

4. Turn on the hook.

Use the HookEnabled property to turn on the windows hooks. Note that this property has no effect on the keyboard hook subsystem of the dwshk80.ocx control.

5. Add your Event Code.

Remember that each type of hook uses a specific event. If you add your code to the wrong event, your code will not execute even if everything else is set up correctly. Look at the on-line reference for the HookType property to see which events are associated with each hook type.

DWSHK80.OCX - Use of the Nodef Parameter

The dwshk80.ocx message events include a nodef parameter. This parameter differs somewhat from the way it works with the dwsbc80.ocx control, and it is important to understand these differences.

With dwsbc80.ocx, nodef is an utterly reliable way to discard a message. The default window routine will not be called.

Dwshk80.ocx uses a different technology to intercept windows messages. Setting the nodef parameter typically prevents all further hooks from being called for the specified message - however, it is not always the case that your dwshk80.ocx control is the first control in the chain. This means that other hooks or tools may have processed the message first, and that some internal Windows operations may have already taken place. In addition, preventing further hooks does not always seem to prevent the message from being fired.

For these reasons, setting nodef to True for the dwshk.ocx control is not recommended and should only be done after careful experimentation.

Depending on the type of hook, you may be able to discard a message by setting the **nodef** property to True and setting the message number to zero.

However, there is a safe way to discard messages when you need to do so by using dwshk80.ocx in conjunction with dwsbc80.ocx. Dwshk80.ocx always receives messages before they are sent to the actual window function. This means that during processing of the message, it is possible to subclass the window using the dwsbc80.ocx control. You can then discard the message via the dwsbc80.ocx control by setting its nodef parameter to True during the message event processing.

The dwGenericHook Object

The dwGenericHook object is part of the dwshvb8.dll component which is written in Visual Basic.

Windows hooks with this component are designed to be implemented in layers. dwGenericHook is the lowest level implementation, handling most of the standard hook types. It is expected that you will build higher level objects that use dwGenericHook in order to provide easy to use interfaces for individual hook types. An example of this is included with the dwPretranslate object.

This component is designed to hook your own application only. This is because most system hooks work by loading the DLL into the process space of each application being hooked. This requires that the DLL be safe to map into other processes spaces, and that you implement a cross-process communication mechanism. Visual Basic does not reliably support either of these. For system and cross process hooks, you should use the SpyWorks dwshk.ocx control.

The steps for using this object are as follows:

1. Add a reference to the "Desaware SpyWorks 8.0 VB Subclasser" object dwshvb8.dll using the Project-References menu.
2. Dimension a dwGenericHook object with events as follows:

Dim WithEvents myobject As dwGenericHook

3. During form load or before using the object, create the object as follows:

Set myobject = New dwGenericHook

4. Use the **HookType** property to set the desired type of hook.
5. Set the **Enabled** property to True.
6. Add code to the **myobject_HookProc** event for the hook events that you are intercepting.

The dwPretranslate Object

The dwPretranslate object provides a functionality known as message pre-translation for use primarily with ActiveX controls implemented with Visual Basic 6.0. This object is based on the dwGenericHook object and provides a good example of how to extend that interface to provide easily used hook functionality.

Message pre-translation is typically used for cases where an ActiveX control must process certain messages that are normally intercepted by the container. A classic example of this is the tab key, which is used by Visual Basic to tab between controls. The dwPretranslate object can intercept the tab key and allow your control to add its own tab key based functionality.

The dwPretranslate object is designed to translate messages destined for a single window only - typically the control's window.

The steps for using this object are as follows:

1. Add a reference to the "Desaware SpyWorks 8.0 VB Subclasser" object dwshvb8.dll using the Project-References menu.
2. Dimension a dwPretranslate object with events as follows:
Dim WithEvents myobject As dwPretranslate
3. During form load or before using the object, create the object as follows:
Set myobject = New dwPretranslate
4. Set the **ExtendedEvent** and **KeyMessagesOnly** properties.
5. Set the **hwnd** property to the handle of your ActiveX control's window (typically the **UserControl.hWnd** property).
6. Add code to the **myobject_PreTranslateMessage** or **myobject_PreTranslateMessage2** events for the messages that you are intercepting.

Hook Examples

This example demonstrates how to detect when the user has clicked the right mouse button over any form or control in your application.

1. Add the Desaware dwshk80 v8 Hook Control component to your project. Add a WinHook control to Form1.
2. Select the WinHook control's **Messages** property to display the Select Messages form and add the WM_RBUTTONDOWN message to the Selected Messages list. We highly recommend *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"* as a reference guide to the Windows messages.
3. Set the WinHook control's **Monitor** property to "4 – This Task".
4. In Form_**Load** event of Form1, set the WinHook control's **HookEnabled** property to True.
5. Attach the following VB code to the WinHook control's **WndMessage** event (since we are using the default HookType – GetMessage, the **WndMessage** event is triggered when messages are detected).

```
' If you are detecting more than one Windows message, you want  
' to compare the msg parameter here to determine which  
' message you received. Since we are only detecting a single  
' message, we can skip this step.
```

```
Debug.Print "User right clicked on " & Hex$(wnd)
```

- This example demonstrates how to monitor the entire system to determine application switching.
1. Add the Desaware dwshk80 v8 Hook Control component to your project. Add a WinHook control to Form1.
 2. Select the WinHook control's **Messages** property Messages to display the Select Messages form.
 3. Add the WM_ACTIVATEAPP message from the Standard message group to the Selected Messages list. We highly recommend *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"* as a reference guide to the Windows messages.
 4. Set the WinHook control's **HookType** property to "4 – WH_CALLWNDPROC".
 5. Set the WinHook control's **Monitor** property to "6 – Entire System".
 6. In **Form_Load** event of Form1, set the WinHook control's **HookEnabled** property to True.
 7. Attach the following VB code to the WinHook control's **WndMessage** event (since we are using the HookType – CallWndProc, the **WndMessage** event is triggered when messages are detected).

```
' If you are detecting more than one Windows message, you want  
' to compare the msg parameter here to determine which  
' message you received. In our case, we just want to detect  
' when an application has been switched.
```

```
Static currentprocessid As Long  
Dim threadid As Long, processid As Long
```

```

If wp Then
    threadid = GetWindowThreadProcessId(wnd, processid)
    If processid <> currentprocessid Then
        currentprocessid = processid
        Debug.Print "Active process changed to " &
            Hex$(currentprocessid)
    End If

    Debug.Print Hex$(wnd) & " window activated, thread " & Hex$(lp) & " is
        de-activated"
Else
    Debug.Print Hex$(wnd) & " window de-activated, thread " &
        Hex$(lp) & " activated"
End If

```

- This example demonstrates how to display the mouse coordinates (in screen coordinates) where ever the mouse moves over your application.
1. Add the Desaware dwshk80 v8 Hook Control component to your project. Add a WinHook control to Form1.
 2. Select the WinHook control's **Messages** property to display the Select Messages form.
 3. Add the WM_MOUSEMOVE message from the Mouse message group to the Selected Messages list. We highly recommend *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"* as a reference guide to the Windows messages.
 4. Set the WinHook control's **HookType** property to "1 – WH_MOUSE".
 5. Set the WinHook control's **Monitor** property to "4 – This Task".
 6. In **Form_Load** event of Form1, set the WinHook control's **HookEnabled** property to True.
 7. Attach the following VB code to the WinHook control's **MouseProc** event (since we are using the HookType – Mouse, the **MouseProc** event is triggered when messages are detected).

```

' If you are detecting more than one Windows message, you
' want to compare the msg parameter here to determine which
' message you received. In our case, we just want to detect when
' the user initiates any mouse activity.

```

```

Debug.Print "Mouse position in screen coordinates: x =
" & X & ",
y = " & Y

```

- This example demonstrates how to monitor the entire system for any type of mouse activity.
1. Add the Desaware dwshk80 v8 Hook Control component to your project. Add a WinHook control to Form1.
 2. Select the WinHook control's **Messages** property to display the Select Messages form.

3. Add all of the messages from the Mouse message group to the Selected Messages list. We highly recommend *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"* as a reference guide to the Windows messages.
4. Set the WinHook control's **HookType** property to "1 – WH_MOUSE".
5. Set the WinHook control's **Monitor** property to "6 – Entire System".
6. In **Form_Load** event of Form1, set the WinHook control's **HookEnabled** property to True.
7. Attach the following VB code to the WinHook control's **MouseProc** event (since we are using the HookType – Mouse, the **MouseProc** event is triggered when messages are detected).

```
' If you are detecting more than one Windows message, you want
' to compare the msg parameter here to determine which
' message you received. In our case, we just want to detect
' when the user initiates any mouse activity.
```

```
Debug.Print "Mouse activity detected, reset count down for screen saver."
```

- This example demonstrates how to detect the Enter key and substitute the Tab key in its place. This is useful if you have a Default command button on a form but wish to allow the user to use either the Enter key or Tab key to move to another control (e.g. a different data field).

1. Add the Desaware dwshk80 v8 Hook Control component to your project.
2. Add a WinHook control to Form1.
3. Select the Enter key from the Select Keys form and hit the Add button.
4. Set the **KeyboardHook** property to "1 – This Task" when appropriate.
5. Attach the following VB code to the WinHook control's **KeyDownHook** event:

```
SendKeys "{TAB}" ' substitute Tab key
discard = True    ' discard Enter key
```

- This example demonstrates how to detect the Control+F2 and Control+F3 keys and run a subroutine. This is useful if you want to create global hot keys that the user can hit at any time (even when your application is not active or is hidden).

1. Add the Desaware dwshk80 v8 Hook Control component to your project.
2. Add a WinHook control to Form1.
3. Select the F2 key from the Select Keys form, check the Control check box and hit the Add button. Select the F3 key, check the Control check box and hit the Add button.
4. Set the **KeyboardHook** property to "2 – Entire System" when appropriate.
5. Attach the following VB code to the WinHook control's **KeyDownHook** event:

```
Select Case KeyCode
    Case vbKeyF2: Call SpecialSubroutineForCtrlF2()
```

```
Case vbKeyF3: Call SpecialSubroutineForCtrlF3()  
End Select
```

```
' This next line is optional and depends on whether you want to  
' disable the Ctrl+F2 or Ctrl+F3 hot keys that may be set by other  
' applications.  
discard = True
```

For Further Information on Hooks

Your best source of information on hooks is the Win32 SDK documentation for the SetWindowsHookEx API function. This is available on the MSDN library CD-ROM.

Many SpyWorks examples demonstrate use of hooks. Please Refer to the on-line reference for a complete list of properties and events for the dwshk.ocx controls and the dwGenericHook and dwPretranslate objects of the dwshvb8.dll component.

SpyWorks Concepts: Private Windows

One of the nice things about Visual Basic is that it manages all aspects of window creation, management and behavior for you. You don't need to worry about defining the window classes for forms and controls, creating the windows, and developing a windows function. All of work is done for you by default. Occasionally you might need to use a subclasser to intercept a particular message, but by and large, Visual Basic provides a complete and satisfactory solution.

But there may be occasions where you actually want to define your own window class and completely define the behavior of windows in that class even to the point of implementing your own window function. To do so you will first want to learn a bit more about window classes and styles, a subject that is covered in Chapter 2 of *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"*. Next, you can use the class definition and window creation API functions directly and use a standard module function with the AddressOf operator to define your own window function for the class. But why bother when SpyWorks has done most of the work for you in the dwshvb8.dll component? Besides, when you use this component to manage your private windows, you don't have to worry about interference with the Visual Basic development environment. This approach does not prevent you from entering break mode and stepping through your application - even with the new class window function code that you create.

The dwPrivateWindow Object

The dwPrivateWindow object is designed to allow you to create and manage your own windows. This feature is especially useful with ActiveX controls, as it allows you to create standard windows that can be created and destroyed as needed, eliminating the problems relating to the inability to change design time only properties of constituent controls. An in depth demonstration of this object can be found in the book *"Dan Appleman's Developing ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed"*.¹

There are two approaches to creating private windows using this object. You can create a standard window such as a LISTBOX, EDIT, COMBOBOX or even common control class, relying on the standard class function implementation to manage the window. You can then use a dwSubClass object to subclass the window's container to intercept notification messages from the window.

A second approach involves creating your own window class and implementing your own standard class function. This approach is more complex but provides total flexibility.

The dwPrivateWindow object is also used to implement delayed message functionality as defined by the **PostEvent** property and **DelayedEvent** event of the dwsbc80.ocx and dwshk80.ocx controls.

The steps for using this object are as follows:

1. Add a reference to the "Desaware SpyWorks 8.0 VB Subclasser" object dwshvb8.dll using the Project-References menu.
2. `Dimension a dwPrivateWindow object with events as follows:

Dim WithEvents myobject As dwPrivateWindow

¹ See <http://www.desaware.com/products/books/com/devcom/index.aspx>

3. During form load or before using the object, create the object as follows:

Set myobject = New dwPrivateWindow

4. Use the RegisterClass method to define a window class if you do not want to use one of the standard window classes or the object defined user window class.
5. Use the CreateWindowEx function to create the window.
6. If you defined your own window class, add code to the **myobject_WndMessage** event to process window messages for windows of the class.
7. If you are using a standard window, use a dwSubClass object to subclass the window's container and watch for WM_COMMAND messages (notification messages) from the window.
8. Before your application ends, be sure to use the DestroyWindow method to destroy the window, then release the dwPrivateWindow object.

SpyWorks Concepts: dwEasy - A Multifunction Control

Subclassing is an extremely powerful technology, but it takes quite a bit of effort and knowledge to perform some advanced tasks. We've collected some of the most useful of these tasks into one control which pre-packages solutions to common problems.

dweasy80.ocx is designed to provide partially pre-packaged solutions to some common Visual Basic 6.0 programming tasks. You will find that even though these are "canned" solutions in a sense, much of the interior workings of these solutions are exposed to make customization easy.

These solutions include:

- Global mouse position tracking.
- Determining update area during paint.
- Creating "rollup" forms and controls.
- Adding scrollbars to forms and controls.
- System menu commands.
- Adding file drop capability to your application.
- Implement and dereference Windows shortcuts using the dwShellLink object (obtained from dweasy80.ocx).

Using dweasy80.ocx is as easy as placing the control on the form or container control that you want to modify.

Placing the DWEASY80.OCX Control

All Visual Basic forms act as "containers" for controls. Many controls can also act as containers (these include picture controls, frame controls, and some third party controls). While some of the features of the dweasy.ocx control (such as Mouse Tracking) do not depend on the placement of the control, some (such as rollups and update area support) work directly on the container of the dweasy80.ocx control.

The most common mistake made by VB beginners is to confuse controls that are overlapping each other and controls that are contained within each other.

To make controls overlap, you draw the control on the form and move it over another control. The Zorder determines which control is displayed on top of the other.

To place a control in a container you must draw it directly on the container. To tell if a control is contained in another, simply drag the container at design time. All controls that are within it will be dragged as well.

At runtime you can switch the container for a control using the SetParent API function - however this can lead to side effects and is not recommended with the dweasy.ocx control.

DWEASY.OCX - Mouse Tracking

One of the most common tasks that people want to do is updating a status bar based on the mouse location on a form. There have been several approaches to this. One is placing code in each **MouseMove** event, using subclassing to detect MouseMove for controls that don't have it. Another is to use a timer. Another is to detect the WM_SETCURSOR message for a form (not reliable if a controls sets its own cursor, and it doesn't work for graphical controls).

Windows mouse hooks provide a mechanism for monitoring the mouse events on a system wide basis. Obviously, it would be possible to use the dwshk.ocx control to monitor the global mouse movement, check for the window under the mouse and update a status bar based on the current position. Of course, the Visual Basic solution for this would be somewhat inefficient, so dweasy.ocx implements this in low level code, providing 99.999% reliable and efficient **MouseEnter** and **MouseExit** events that work for every control on a form except for some graphical controls. (Graphical control support works for most controls placed directly on a form, however it is not reliable with multiple levels of nesting.) This feature is enabled using the **MouseTransit** property. We recommend that you set the **MouseTransit** property to track windows only unless absolutely necessary, as tracking graphical controls is relatively inefficient.

Invisible windows and controls are not detected by mouse tracking.

The drop down portion of a drop down list box is not detected by mouse tracking (this is due to an internal Windows quirk by which this drop down listbox is owned by the desktop window instead of the combo box itself).

The control includes a **MenuSelect** event that allows you to monitor menu selection state as well.

Relevant Properties:

DetectDisabled Property	MDIMenuDetect Property
MouseTransit Property	MouseTransitNC Property
TransitTag Property	

DWEASY80.OCX - Determining Update Area During Paint

One of the limitations of the Visual Basic **Paint** event is that there is no way to determine during the event which part of the window actually needs to be updated. When images are complex, this can lead to unnecessary drawing.

The `dweasy80.ocx` control records the update area which can be read during the **Paint** event using the **UpdateLeft**, **UpdateRight**, **UpdateTop**, **UpdateBottom** properties. These properties contain the Left, Right, Top and Bottom coordinates of the update area in the current container coordinates.

The `dweasy80.ocx` control should be placed directly on the form or control for which you want to determine the update area.

Relevant properties:

Update properties **UpdateLeft**, **UpdateRight**
 UpdateTop, **UpdateBottom**

DWEASY.OCX - Tiny Captions and Rollup Windows

Two variations on standard windows have become more common in many applications. One is a window that uses a "tiny" caption. One example of this is the Visual Basic control toolbar. Another is the kind of "rollup" control used in programs like Corel Draw (in which it looks like a normal window until you roll it up, at which time only the caption is displayed).

Rather than create special controls that perform these operations, `dweasy.ocx` modifies the standard picture control (or most other controls that can act as a container). Simply place the `dweasy80.ocx` control on the form or control that you wish to modify.

Notes on working with Tiny Captions and Rollups

In accordance with the SpyWorks philosophy of providing tools rather than just solutions, we have not completely characterized this control under every possible configuration. Instead we chose to expose some of the internal working of the control through events and provide you with the capability to modify the action of the control if you wish.

We would be very interested in hearing suggestions on how to extend or modify this capability. While we don't guarantee to make the changes you want, if it sounds good we may be able to incorporate the new feature and send it to you within a very short period of time.

Additional Notes:

Forms that are modified by setting the **CaptionStyle** property to a non-zero value may not have visible menus. Form menus can be created as invisible for use with the Popup Menu function.

Double clicking on the control box of resized or rollup containers has no default behavior.

The "tiny caption" and "rollup" features of `dweasy.ocx` are not designed to be used with MDI child forms or MDI forms. It will produce some interesting results.

When using rollups with forms, you should keep the **CaptionHeight** property greater or equal to 100. This is because Windows sets the minimum size for a window which overrides the attempt of `dweasy.ocx` to reduce the height of the window below that of a standard caption. If you use a **CaptionHeight** smaller than 100, a small portion of the form will remain visible under the caption. This limitation does not apply to rollups made from containers that are controls on a form.

No Min or Max button will appear when the **CaptionStyle** is '1 - Resized' for non-form containers. This is because the concept of "minimize" and "maximize" does not really apply to child windows.

Relevant Properties/Events:

CaptionHeight Property	CaptionStyle Property
ForceActive Property	ForceCtlBox Property
ForceOutline Property	ForceTitle Property
MinMax3D Property	RolledUp Property
ShowInTaskbar Property	StateButton Event
SysMenuRequest Event	

DWEASY80.OCX - Adding Scrollbars to Forms and Controls

Visual Basic does not provide support for the scrollbars that are potentially built into most windows. dweasy80.ocx not only allows you to turn on and use those scrollbars, but provides some automated scrolling capabilities as well. The scrolling capability can range from user defined control of the scrollbars, to "terminal" style scrolling support, to true virtual form or control capabilities.

The scrollbars properties and events are in most cases identical to the standard Visual Basic scrollbar control properties and events except that they are preceded by a 'V' or 'H' character to indicate the vertical or horizontal scrollbar.

Simply place the dweasy.ocx control on a form or container control, and use the **ScrollBars** property to turn on one or both scrollbars.

This control is not designed for use with forms and picture controls that have their AutoRedraw property set to True.

The scrolling properties use integer values. Be sure to consider the possibility of overflows towards the high and low ends of the scrolling range.

Automatic scrolling with this control is done entirely in pixels. You can perform your own coordinate transformations to twips if you wish.

The dwshvb8.dll includes the dwScrollBar component which can also be used to turn on scrollbars for windows in an application. It does not include the virtual form capabilities of dwEasy, but it is written in Visual Basic and is idea for turning on the scrollbars for a Visual Basic User Control object.

Relevant Properties/Events:

Change, Hchange Event	ScrollBars Property
VScroll, Hscroll Event	ScrollViewport Property
VSmallChange, HSmallChange Properties	ScrollUpdate Property
VValue, Hvalue Properties	ScrollWindow Property
VLargeChange, HLargeChange Properties	VMax, HMax Properties
VMin, HMin Properties	

DWEASY80.OCX - Other Functions**System Menu Support**

Allows you to easily detect system menu commands. Refer to the **SystemCommand** event for additional details.

DWEASY80.OCX - Windows Explorer/File Manager Drag-Drop Commands

Visual Basic 5.0 (and later) includes support for OLE drag and drop which provides effective drag-drop capability across applications. dwEasy does include file drag drop support for use with Visual Basic 4.0 that enables your form or container to receive dragged files from File Manager, Explorer, or other applications.

Relevant Properties/Events:

DetectDropped Property	DroppedFile Property
FileDropHwnd Property	FilesDropped Event

dwShellLink - Shell Link Support

The dweasy80.ocx control provides support for the IShellLink interface under Windows 95/98/ME and Windows NT (version 4.0 and higher). This interface is used to implement shortcuts to files. Each shortcut can have its own working directory, calling arguments and icon, providing an efficient way to reference files in different ways without having multiple copies of the file. Windows also supports automatic link resolution that can often find a file even after it has been moved or renamed.

The most common use of the **dwShellLink** object is with file drag/drop support. When a shortcut is dropped onto a window that has file drag/drop enabled (using the dwEasy **DetectDropped** property), the control receives the name of the shortcut file (with the extension .LNK) instead of the file that is referenced by the shortcut. dwEasy's IShellLink support makes it possible to resolve the link to determine to which file it refers. It also allows you to create and modify links.

Obtaining a dwShellLink Object

dwEasy implements the IShellLink interface by exposing an object of type dwShellLink. You can obtain a dwShellLink object using the dwEasy GetShellLink method. This method is defined as follows:

1. GetShellLink(FileName as String, AllowPrompt as Boolean)

The **FileName** property allows you to immediately initialize the dwShellLink object to a shortcut file. The **AllowPrompt** property determines whether Windows is allowed to bring up a dialog box to allow you to resolve the link if it has been broken (because the target file has been moved or renamed). If you set AllowPrompt to False, and Windows is unable to resolve the link automatically, a run-time error will occur. If you try using this function in an operating system without shell links, you will also get a run-time error. You can pass an empty string as the FileName to obtain an uninitialized dwShellLink object. You'll need to call the LoadLinkFile method to associate the object with a link file if you wish to use an existing link file.

The general sequence for using the dwShellLink object is as follows:

2. Obtain an object as follows:

```
Dim myobject as dwShellLink
Set myobject = sbcEasy1.GetShellLink("", False) ' (or specify the file here)
(optional) Load a link file into the object using the LoadLinkFile object.
(optional) Set new values into the link file object.
(optional) Save the link file using the SaveLinkFile object.
```

Notes on Working with Shell Links

Some of the information associated with shell links can be seen by viewing the properties for the link under the Windows Explorer. You should note, however, that some of the information (including the **ShowCmd** property and ShowCmd associated icon) is cached by Explorer and thus changes made through the dwShellLink object will not be reflected immediately when these properties are viewed in Explorer.

dwEasy Examples:

- The following demonstrates how to display status help for menus selected in a form.
 1. Create a new project with a single form and a label control.
 2. Add a menu to the form and name it "File".
 3. Add 3 sub menus to "File" and name them "New", "Open" and "Save".
 4. Add a dwEasy control to the form. The dwEasy control automatically detects menu selections on its container.
 5. Attach the following code to the dwEasy's **MenuSelect** event:
' You can also do the comparison based on the menu's position
' by comparing the position parameter, but do this only if the
' menus positions will not change.

```
Select Case MenuName
    Case "New": Label1.Caption = "Creates a new document."
    Case "Open": Label1.Caption = "Opens an existing
document."
    Case "Save": Label1.Caption = "Saves the current
document."
End Select
```

- The following demonstrates how to display status help for menus selected from a MDI form.
 1. Create a new project with a single MDI form and a label control.
 2. Add two MDI child forms.
 3. Add a menu to the first MDI child form and name it "File".
 4. Add 3 sub menus to "File" and name them "New", "Open" and "Save".
 5. Add a menu to the second MDI child form and name it "Edit", then add 3 sub menus to "Edit" and name them "Cut", "Copy" and "Paste".
 6. Add a dwEasy control to the MDI form. The dwEasy control automatically detects menu selections on its container.
 7. Attach the following code to the **Form_Load** event of the MDI form:
Form1.Show
Form2.Show
 8. Attach the following code to the dwEasy's **MenuSelect** event:

' You can also do the comparison based on the menu's position by
' comparing the position parameter, but do this only if the menus
' positions will not change.

```
Select Case MenuName
    Case "New": Label1.Caption = "Creates a new document."
    Case "Open": Label1.Caption = "Opens an existing document."
    Case "Save": Label1.Caption = "Saves the current document."

    Case "Cut": Label1.Caption = "Cut the selected data and place
    It on the clipboard."
    Case "Copy": Label1.Caption = "Copy the selected data and
    place it on the clipboard."
    Case "Paste": Label1.Caption = "Paste the data from the
    clipboard to the insertion point."
End Select
```

- The following demonstrates how to display status help for controls that the mouse is currently over in your form.

1. Create a new project with a single form and a label control.
2. Add a command button, a picture box, and an image control to the form.
3. Add a dwEasy control to the form.
4. Set the dwEasy control's **MouseTransit** property to "2 – Track all controls" (this will allow dwEasy to detect the image control, if you do not need to track graphical controls, set the **MouseTransit** property to "1 – Track windows" for better efficiency).
5. Attach the following code to the dwEasy's **MouseEnter** event:

```
Select Case ControlDesc
    Case "Label1": Label1.Caption = "Mouse is over the label"
    Case "Command1": Label1.Caption = "Mouse is over the
    command button"
    Case "Picture1": Label1.Caption = "Mouse is over the picture
    box"
    Case "Image1": Label1.Caption = "Mouse is over the image
    control"
End Select
```

6. Attach the following code to the dwEasy's **MouseExit** event:

```
Label1.Caption = ""
```


SpyWorks Concepts: dwShengine80.dll Function Library

IMPORTANT NOTE: dwshengine80.dll is a replacement for dwspy36.dll for SpyWorks version 8.0 and later. All functions in dwspy36.dll are included in dwshengine80.dll and are called in the same way. The term dwspy3x.dll is intended to refer to both DLLs. The dwshengine80.dll component contains the new version 8.0 anti-spyware technology. Like dwspy36.dll it is built using ATL for minimum dependencies. An older edition of this component, dwspy32.dll, was compiled using MFC.

The dwshengine80.dll dynamic link library contains a selection of functions that have proven useful over the years. They are intended to fill holes in the capabilities of Visual Basic.

As it has evolved, many of the functions in dwshengine80.dll have been rendered superfluous as Microsoft added equivalent functionality to Visual Basic. In addition, some capabilities that were supported in 16 bit editions of Visual Basic could not be implemented in the 32 bit edition due to significant changes in the underlying architecture.

These components are also used by the Desaware ActiveX controls to perform their tasks, and thus must be distributed (dwshengine80.dll with dwshbc80.ocx; dwshk80.ocx; and dweasy80.ocx; dwspy36.dll with dwshbc36.ocx; dwshk36.ocx; and dweasy36.ocx; dwspy32.dll with dwshbc32.ocx; dwshk32.ocx; dweasy32.ocx and dwcbk32.ocx; dwspydll.dll with the 16 bit OCX and VBX controls).

The dwshengine80.dll functionality falls into a number of categories as follows:

Data and Memory Access

User Defined Type Packing Functions

Miscellaneous Functions

DATA AND MEMORY ACCESS

dwspydll.dll and dwspy3x.dll contain a selection of functions designed to help ease the process of working with API functions, especially with regards to manipulating text strings and pointers.

dwspy3x.dll also includes functions to transfer data to and from other process spaces. This can be critically important when subclassing other applications.

Function Name	Description
dwDWORDto2Integers, dwDWORDto4Bytes	Breaks a 32 bit long variable into integers or bytes.
dwCopyData, dwCopyDataBynum	Copies a block of memory.
dwSwapBytes, dwSwapWords	Swaps bytes within an integer or integers within a long variable.

Function Name	Description
dwGetAddressFor ...	Obtains the address of a variable.
dwGetAddressForLPSTR	Obtains the address of a string.
dwGetStringFromLPSTR, dwGetStringFromPointer	Converts a pointer containing a null terminated string into a VB string.
dwHugeOffset	Calculates an address offset (16 bit).
dwPOINTSToLong	Converts a POINTS structure into a long

	variable.
dwMakeROP4	Creates an ROP4 structure.
dwGetStringFrom2NullBuffer	Extracts a string from a list of strings separated by null characters and terminated by two null characters.
dwSpy3x.dll - dwXCopyDataTo, dwXCopyDataFrom	Copies data between processes.
dwSpy3x.dll - dwXCopyAnsiStringFrom, dwXCopyUnicodeStringFrom	Retrieves a string from another processes memory space.
dwXAllocateDataFrom, dwXFreeDataFrom	Allocates data space in the memory space of another process.
dwIsValidName	Determines if a name is a valid method or property name for an ActiveX automation component.

User Defined Type (UDT) Packing Functions

Consider the following C structure:

```
struct mystruct
{
    a as short
    b as long
    c as short
    d as short
}
```

An API function expecting this structure as a parameter would expect to see a total of 10 bytes. The first two bytes containing integer 'a', followed by four bytes containing variable 'b', followed by 'c' and 'd' respectively.

You cannot pass this type of structure to a DLL using Visual Basic 4, 5 or 6.

Why is this? Because Visual Basic enforces certain alignment rules in user defined structures. Within Visual Basic itself, all variables start on a 32 bit (4 byte) boundary. When VB passes a structure to a DLL it packs it part of the way - so that each variable in the structure starts at its natural boundary. This means that bytes start anywhere, integers (16 bit variables) start at a 16 bit boundary (even byte addresses) and long variables start at 32 bit boundaries (byte addresses evenly divisible by 4). In this case the DLL would receive a structure in which the first two bytes contain integer 'a', followed by 2 unused bytes, followed by four bytes containing variable 'b'. Variables 'c' and 'd' follow immediately, since they fall on even addresses. The resulting structure ends up containing 12 bytes - not the 10 expected, and variables b, c and d are in the wrong locations.

The good news is that this occurs rarely when using the Win32 API - most API structures are designed so that all of their variables fall on their natural boundaries. Unfortunately, some of Microsoft's programmers don't always follow their own programming standards, and some API structures (such as a couple used in common dialogs) require single byte packing in order to work.

Until now this required careful work-arounds on the part of Visual Basic programmers. It was necessary to either divide long variables into two integers and recombine them later, or to carefully copy a Visual Basic user defined structure into a byte array (moving variables as necessary), passing the address of the array to the DLL, then copying the data back into the user defined structure when done.

SpyWorks includes a number of functions to automate this process, making it easy to load a memory buffer with a packed version of a user defined structure, and to unpack it when done.

Relevant Functions:

dwFreeUDT	dwPackedUDTSize
dwPackUDT	dwRegisterUDT
dwUnpackUDT	

Please refer to the on-line Help file for further information.

Miscellaneous Functions

These are functions that do not fit conveniently into any of the other categories.

Function Name	Description
Control Analysis Functions	Used to analyze 16 bit custom controls. Not implemented in 32 bit.
dwFindFiledwSpy	Searches for a file on the disk starting with the directory containing the executable file for the current process.
dwSubtractFileTimes	This function subtracts the value in one Win32 API FILETIME structure from another. This can be useful for performing relative timing using the GetProcess-Times function.
dwXGetModuleFileName	This function retrieves the executable file name (including path) for the specified process as long as the process has a window.
GetSpyWorksVersion	Returns a string containing the current version of SpyWorks.
Port I/O functions	These functions allow direct port access under Windows 95/98. Windows NT's security system prevents them from working under Windows NT.
Printer Driver Function Access	These functions are used under 16 bit Windows only.

dwSpy3x Examples

Please refer to the on-line Help file for these examples.

SpyWorks Concepts: Exporting Functions

SpyWorks 5 introduced a new technology called Dynamic Export Technology which allows you to export functions from any ActiveX DLL including ActiveX controls. This technology provides outstanding performance and ease of use, yet avoids modifying your compiled DLL or OCX in any way, eliminating any possibility of build errors or the chance that your component may be corrupted.

Version 7 extended Dynamic Export Technology for use with Visual Basic .NET and C#, allowing you to export functions from DLL assemblies written in either of these languages.

Note: If you are upgrading from SpyWorks 5.0 Professional Edition to a later version, be sure to upgrade your projects as follows:

- Your ActiveX DLL must be marked for apartment model multithreading if you expect your DLL to be used with multithreaded clients (not applicable for .NET).
- You must install Visual Studio Service Pack #2 or later (for Visual Basic 5.0) before compiling your ActiveX DLL (not applicable for .NET).

The documentation that follows will cover these issues in more detail.

What are Exported Functions?

Every Dynamic Link Library (DLL) consists of a collection of functions.

Those DLLs that make up Windows contain many hundreds of functions called Windows API functions (API = Application Programming Interface). You can access those functions from Visual Basic using the Declare statement. In order for a function to be accessible using the Declare statement, it must be "exported" - this means that the name of the function and its location in the DLL is made public by the DLL.

There is another way for DLLs to expose functions - by using OLE interfaces (also known as ActiveX interfaces). These DLL's export a single function called `DllGetClassObject` which can be used by the system to obtain a COM object - an object compatible with the Common Object Model. An in depth look at using these types of objects can be found in the book "Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed".² You'll need to understand the fundamentals of creating these ActiveX DLLs in order to use the dynamic exporting capability of SpyWorks.

.NET assemblies expose functions through yet another mechanism, publishing public objects in the assembly's manifest. Objects exposed through .NET assemblies are created and managed by the Common Language Runtime (CLR).

Visual Basic 5.0 and 6.0, Visual Basic .NET and C# do not allow you to export functions, yet this can be an important feature. If you can export functions, you can create your own export function libraries that other applications can access, whether they are written in Visual Basic (in which case you use the Declare statement), or written in other languages.

² See <http://www.desaware.com/products/books/com/index.aspx>

Some operating system features will only work with dynamic link libraries that export functions. For example, you must be able to export functions to create control panel extension applets. Also, NT services (which typically use control panel applets) benefit from this capability. Lastly, certain services and applications will only work with dynamic link libraries that export functions. For example: the Internet Service API (ISAPI) requires this capability.

Exporting functions from a Visual Basic 6.0 DLL or ActiveX control is a two step process.

1. Add a dwExporter class to your ActiveX DLL or control.
2. Use the dwExUtil program to create an alias DLL which loads your DLL and links the functions.

Similarly, exporting functions from a Visual Studio .NET DLL is also a two step process.

1. Add the ExportAttribute template file to your project.
2. Use the ExportWizard to create an alias DLL which loads your DLL and links the functions.

But first, we recommend you read a little bit about how Desaware's Dynamic Export Technology works.

How Dynamic Export Technology Works

Desaware's Dynamic Export Technology works without modifying your ActiveX DLL or control (or .NET assembly) in any way. This has two advantages:

- It eliminates any chance that Desaware's tools might corrupt your component file.
- It eliminates any chance that you might forget a step after recompiling your component.

The following figure illustrates the operation of the SpyWorks exporting feature.

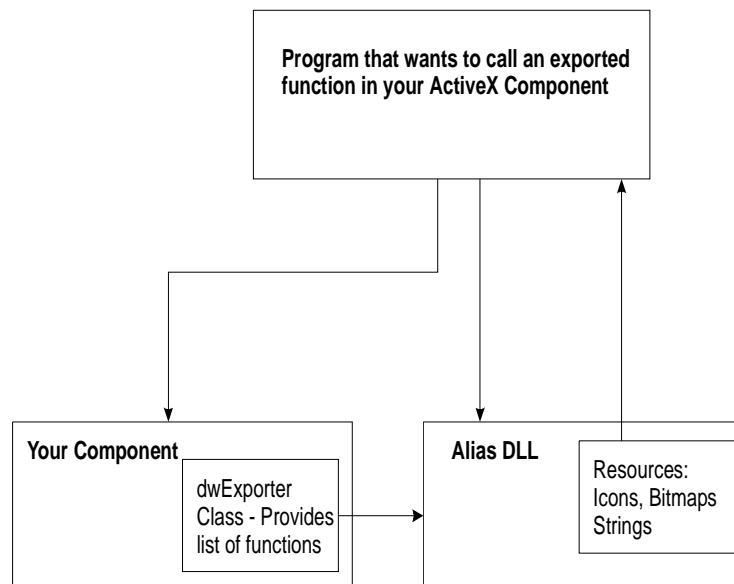



Figure 5
SpyWorks Exporting Feature

First, you'll implement the dwExporter class in your component in VB 5 or 6. This class has a number of methods that will be called by SpyWorks when your component loads in order to find out about the functions that it wants to export. For Visual Studio .NET, you define a Public class called Exports in your project.

The real work is done by a separate DLL called an Alias DLL. For VB 6, this DLL is created by the dwExUtil utility program. The utility program requires that you specify the name of your component and any string, bitmap or icon resources that you wish to include. It then creates the alias DLL with a name that you specify. For Visual Studio .NET, you use the ExportWizard utility program to create the Alias DLL. The ExportWizard requires that you specify the name of your .NET DLL and version resource. The ExportWizard also requires that your .NET DLL exists before creating the Alias DLL.

As far as other programs or the windows system are concerned, your functions are exported by the Alias DLL - thus you will specify the Alias DLL as the  to use in Declare statements or by the system.

But when the other application or the system actually loads the Alias DLL, it will load your component DLL and read its dwExporter class information (VB 6 - for .NET it reads the Exported attribute information) to obtain the list of functions that you wish to export. It will then dynamically link those functions that they may be accessed by the calling application.

From then on, the calling application or system will call the functions in your component directly - completely bypassing the Alias DLL. This approach offers the best performance possible.

For VB6 components, exported functions represent direct entry points into an application, so it is important that certain threading rules be followed if you are exporting functions for use by a multithreaded client. Most problems can be avoided by simply marking your component for Apartment Model multithreading.

For .NET assemblies, the alias DLL performs the necessary transition from unmanaged code (called by those using the exported function), and your managed assembly. However, all incoming calls will be on the caller's thread. The Alias DLL performs no synchronization, so if your caller is multithreaded, be sure to synchronize incoming calls as needed for your particular application.

Sometimes you may want your component to provide resources such as icons, bitmaps or strings to the calling application. Even though the calling application is calling functions in your ActiveX component, it thinks it is calling functions in the Alias DLL, and that is where it will look for these resources. This is the case with control panel extensions.

For this reason, the dwExUtil program allows you to embed resources into the Alias DLL. It uses a subset of the Desaware Resource Compiler from Desaware's StorageTools to accomplish this.

The dwExporter Class (for VB6)

The first step in exporting functions from your ActiveX DLL or control is to add a dwExporter class object into your application. We've included a template file that will get you started.

Critical first steps: (must be followed exactly as stated):

1. Set the class name to dwExporter (no other name will work).
2. Set the class module's **Instancing** property to 5 - MultiUse.

3. Add a reference to the file dwExport.tlb. You will **not** need to distribute this file with your component.
4. Add the following line to the class module: Implements IdwDynamicExport.

The dwExporter class must implement the following private functions:

- Private Sub IdwDynamicExport_GetFunctionCount(FunctionCount As Long)
- Private Sub IdwDynamicExport_GetFunctionInfo(ByVal FunctionNumber As Long Ordinal As Integer, FunctionName As String, FunctionAddress As Long)
- Private Sub IdwDynamicExport_GetModuleHandle(ModuleHandle As Long)

You will also want to add the following function to the class, if you do not have it already in a public module in your component:

```
Private Function GetAddress(ByVal l As Long) As Long
    GetAddress = l
End Function
```

This function is used to convert the address of a function in a standard module into a long variable.

The following is a review of the three IdwDynamicExport functions:

GetFunctionCount

This function is used by your component to specify the number of functions that you are exporting. Simply set the FunctionCount parameter to the number of functions to export. This example illustrates a component that exports two functions.

```
Private Sub IdwDynamicExport_GetFunctionCount(FunctionCount As Long)
    FunctionCount = 2
End Sub
```

GetModuleHandle

This function is used to let the Alias DLL know your component's module handle. This is necessary for the linking process to work correctly.

This function will always look exactly like this:

```
Private Sub IdwDynamicExport_GetModuleHandle(ModuleHandle As Long)
    ModuleHandle = App.hInstance
End Sub
```

GetFunctionInfo

This function will be called by the Alias DLL once for each function that you have exported. It first calls GetFunctionCount to determine how many functions you are exporting, then goes through the list one by one (starting with function #1).

A typical implementation would be as follows:

```
Private Sub IdwDynamicExport_GetFunctionInfo(ByVal
    FunctionNumber As Long, Ordinal As Integer, FunctionName
    As String, FunctionAddress As Long)
    Select Case FunctionNumber
        Case 1
            Ordinal = 10
            FunctionAddress = GetAddress(AddressOf MyExportedFunction)
            FunctionName = "MyExportedFunction"
```

Case 2

Ordinal = 11

FunctionAddress = GetAddress(AddressOf MyExportedFunction2)

FunctionName = "VariableAddress"

End Select

End Sub

The FunctionNumber parameter is a value from 1 to the value you returned in the GetFunctionCount call specifying the function for which you should return information.

For each exported function you should set the Ordinal, FunctionName and FunctionAddress parameters as follows:

- Set the Ordinal to any 16 bit number other than zero. This is called the function's ordinal value, and you'll generally assign the values sequentially. In most cases the value has no significance, but you should set them in order, and each ordinal value should be unique.
- The FunctionAddress parameter is set to the address of any public subroutine or function in a standard module. The AddressOf operator obtains the address and passes it to the GetAddress function, which returns it as a long variable.
- The FunctionName parameter is set to the name under which you want the function exported. This name need not match the name of the function in the standard module.

You can add new functions to be exported at any time. You can delete them as well, but this may break other applications that use your DLL. All you need to do is adjust the number returned in the GetFunctionCount function, and return the information for the function in the GetFunctionInfo function, then recompile your component.

You should mark your component for Apartment Model threading before compiling the DLL.

You do not need to create a new alias DLL when you change the functions being exported.

The Desaware Export Utility (for VB 6)

The Desaware Export Utility is used to create the alias DLL which provides support for the Dynamic Export Technology. It includes a simple resource compiler which allows you to add string, icon or bitmap resources to the Alias DLL.

Once you create an Alias DLL for a component, you will never need to change it unless you want to add or change the resources in the DLL. You do not need to create a new Alias DLL when you change the list of functions being exported - the Alias DLL creates the export function list dynamically based on the functions specified in the dwExporter class.

The Project Menu is used to create the Alias DLL.

It contains the following commands:

- Set Source Project. Use this command to specify the project name of your component (as set in the Project-Properties dialog box for your component). The Alias DLL will attempt to load the object ProjectName.dwExporter to obtain the exported function list, so if you don't use the correct project name, the exporting operation will fail.

- The Set Alias DLL command. Use this command to specify the name and path for the newly created Alias DLL.
- Set Custom Error Message command. Use this command to specify an error message which will be displayed by the Alias DLL if it cannot successfully load your Visual Basic component. If you do not enter an error message, a default error message will be displayed.
- The Build Alias DLL command. Use this command to create the new alias DLL.

Important:

The file dwExport.dll must be in the same directory as the dwExUtil.exe program. This DLL is the template from which your Alias DLL is created. You should not distribute dwExport.dll.

You can use the other dwExUtil commands to add resources to the Alias DLL.

What are Resources? (for VB 6)

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 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 *"Dan Appleman's 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. Desaware's StorageTools product includes an easy to use resource compiler that is written in Visual Basic and includes full source code. This technology was adapted for the dwExUtil project to add resources directly to your Alias DLL. The dwExUtil program neither creates nor uses .res files, but it is able to use the same .drl and .drc resource files used by the Desaware resource compiler, and is able to create these files.

Adding Resources to the Alias DLL (for VB 6)

There are three types of files that you will need to be concerned about when adding resources to the Alias DLL. The first is a project file or "script" file which contains a list of the resources to use. This utility uses two proprietary script file formats. The first is a list file or .drl file. This file contains a list of bitmap files, icon files and strings in a simple text format. The second is a .drc file which uses OLE structured storage to save the resource information.

The .drl file has the advantage of being very fast and easily editable using any text editor such as Notepad. However, if any of the files in the list are missing, you will not be able to compile the project file (the missing resource will be ignored).

The .drc file stores the entire resource in the resource file along with the original file name and date. This ensures that the project can be recompiled even if the original resource files are missing. It also allows the project to automatically update the .drc file if a more recent version of the file is found. This approach does take up additional space and is somewhat slower when saving the file (load times and compile times are comparable to the .drl files).

The first step in using the dwExUtil resource features is to select a project file, by either opening an existing .drl or .drc file, or by creating a new one. This is done using the File-New or File-Open commands.

You can then add or delete any of the resources for the file. If you make any changes, you will be prompted to save the file when you close the application or try to open a new project file. You can also use the File-Save command to save the project file at any time. The File-Save As command can be used to copy the current project file into a new one.

.drl and .drc files are common to both the 16 and 32 bit versions of an application. In other words, the same file can be loaded under either environment.

You do not need to add resources to an Alias DLL. If you would prefer not to do so, simply ignore all of the menus other than the Project menu of the dwExUtil program.

The dwExUtil Main Form (for VB 6)

The primary form for the Desaware Export Utility is shown below:

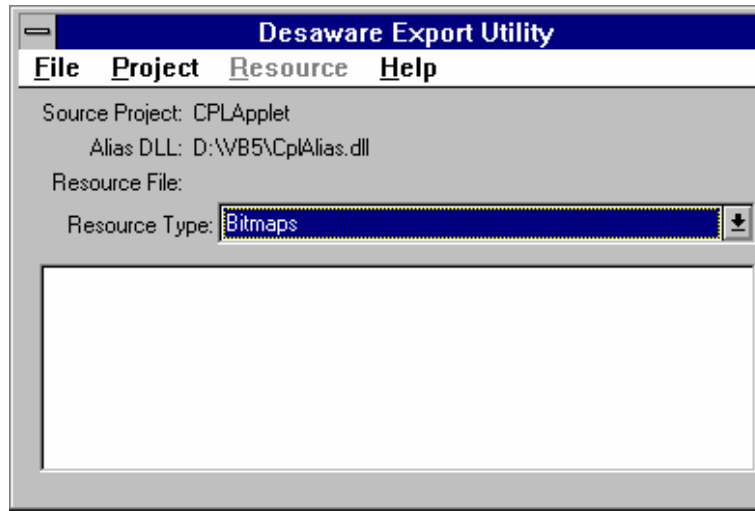


Figure 6

Desaware Export Utility

At any given time the following information is displayed:

Source

Project Field	The project name of the component from which you are exporting functions.
Alias DLL Field	The name and path of the Alias DLL that you will build.
Resource File Field	The current project file (resource script) as selected using the File menu.
Resource Type	A combo box which can be used to select bitmaps, icons or strings.
List Box	How the resources in the project file.

Use the Resource-Add menu to add a new resource. For bitmap and icon resources, this will bring up a file common dialog with which you can select a new bitmap or icon file. After you select a file, you will be asked to specify a numeric resource identifier.

For string resources, this will bring up a string resource editing form which allows you to add a string, edit a string and assign a resource identifier.

The resource identifier should be from 2 to 32768. The resource compiler will prevent you from selecting a resource that is currently in use by another resource of the same type (in other words, you can specify only one bitmap with a given ID, but a bitmap and icon may share the same ID).


If you select an entry in the list box you may use the Resource-Delete command to delete it (or press the Del key). In the case of strings, you can double click on an entry or use the Resource-Edit command to edit an existing string resource.

String resources are more efficient if you keep them in sequential groups (every 16 string identifiers are stored in one resource).

The Exports Class (for Visual Studio .NET)

The first step in exporting functions from your .NET assembly is to add the ExportAttribute class file into your project. We've included a VB and C# template file that you can add to your project. You do not need to make any changes to this file.

Critical first steps: (must be followed exactly as stated):

1. Add a Public Class with the assigned name Exports to your project (no other name will work).
2. Define Public Shared (VB) or public static (C#) functions in the Exports class for functions that are to be exported. 
3. Add the Exported attribute to these functions, define the exported function name and exported ordinal value in the attribute. The function name is set to the name under which you want the function exported. This name need not match the actual name of the function in code.

Set the function's ordinal value to any 16 bit positive number. In most cases the value has no significance, but you should set them in order, and each ordinal value should be unique. You may also specify the 'C' calling convention for the exported function (the default is the standard API calling convention).

<Exported("MyExportFunc", 1)> Public Shared Function...

<Exported("MyExport", 2, CCall:=True)> Public Shared Function...

You can add new functions to be exported at any time. You can delete them as well, but this may break other applications that use your DLL. All you need to do is recompile your project. You do not need to create a new alias DLL when you make changes to the functions being exported.

Currently, byval and byref passing of the following function parameter and return data types are supported:

System.Int32
System.Int16
System.Byte
System.IntPtr

The ExportSample project demonstrates how to use the System.Marshal class to pass other data types (such as strings and structures).

The Export Wizard (for Visual Studio .NET)

The Desaware Export Wizard (ExportWizard) is used to create the Alias DLL which provides support for the Dynamic Export Technology under .NET.

Step	Explanation
Alias DLL File Name	This step is used to specify the path and file name of your alias DLL.
DLL and Assembly Name	This step is used to specify the path and file name of your .NET DLL that includes the function export code. The DLL file's Assembly Name will be retrieved for your verification.
Version Resource	This step allows you to specify a

version resource for the Alias DLL file. At a minimum, the File Version and Company Name must be specified.

Assembly Verification	This step scans the specified Assembly for the required declarations, formatting errors, and other inconsistencies regarding the Assembly.
Compile	This step compiles the specified Alias DLL file.

Once you create an Alias DLL for an assembly, you will never need to change it. You do not need to create a new Alias DLL when you change the list of functions being exported - the Alias DLL creates the export function list dynamically based on the functions specified by the exported attribute.

Testing Exported Functions

Functions can only be exported within the process space of an application. This means that the Alias DLL and your component must be running within the same process space as the calling function.

This means that you will not be able to run your component within Visual Basic while another application calls exported functions in that component. You must compile it into a DLL or OCX first.

The following suggestions should help you with your debugging process:

- Create test routines within your component that call the exported functions in the standard modules directly. You can expose these routines through a debugging object which can be accessed by a standard project that you can load at the same time as your component.
- Use the MessageBox commands, modeless forms and debug monitors. (A separate application that has a public object which has functions to display messages. Your component can access that object and call its functions to display information in much the same way that you would use a Debug statement.) These will help you trace execution and information within your component.

Distributing Your Component

Distributing the Alias DLL for VB 6

You must distribute the Alias DLL with your component. Remember that applications using your component must actually refer to the Alias DLL - it will redirect the exported functions as needed. The Alias DLL need not be in the same directory as your component.

Your component must be registered in order for the Alias DLL to work - it must find the dwExporter object in your component. The Alias DLL itself need not (and in fact cannot) be registered.

Distributing the Alias DLL for .NET

You must distribute the Alias DLL with your .NET assembly. Remember that applications using your assembly must actually refer to the Alias DLL - it will redirect the exported functions as needed. The Alias DLL must be installed in the same folder as your assembly or installed in a shared folder. Your .NET assembly must be installed in the same folder as the calling application, or in the same folder as the Alias DLL.

Warning! Exporting Functions is Dangerous!

When you export a function from a DLL component you are providing a function address which will be called directly by the calling application or the system.

The number of parameters and parameter types of your exported functions must be correct - in other words, they must match exactly what the calling application is using.

If you get it wrong you will almost certainly trigger a memory exception. There is no error checking provided by Visual Basic - this is part of the nature of exporting functions.

So double and triple check those declarations!

Be sure that if you are exporting a function (as compared to a subroutine) that you return the correct data type. Specifying the incorrect data type can also trigger a memory exception.

Be sure that your ActiveX DLL is marked for Apartment Model multithreading if you expect it to be used by multithreaded clients.

.NET assemblies should not throw errors to the caller. The client is most likely not a .NET application and will not be able to handle the errors. Our framework does forward any errors thrown to the calling function, however you should assume that the caller will not handle it and that this will actually result in a memory exception.

It is important that you catch any errors raised in your code. We recommend you return an error value, and possibly call the SetLastError API function to provide additional error information to the caller.

Control Panel Applets and Multithreading Clients in Visual Basic 6

When a client is multithreading several issues come into play:

It makes it more difficult for the Alias DLL to obtain the export information from your DLL, since this operation depends on OLE and Visual Basic's ability to handle multithreaded clients.

The client thread calling the exported functions may not be the same thread in which your DLLs objects are running, meaning that cross-thread marshaling of data will be necessary.

If multiple threads in the client call your exported functions, you may run into race or deadlock conditions, since Visual Basic does not provide synchronization at this level.

This situation also applies to control panel applets under Windows 95/98 and NT 4.0, since they run in different threads under Explorer.

SpyWorks includes a framework for authoring control panel applets that is similar to the technology used to create NT services. As with services, it allows you to test and debug your control panel applets using the Visual Basic IDE. This implementation however, requires that you create your control panel applet on an NT 4.0 or later based operating system. Your control panel applet may be deployed on Windows 95/98/ME and Windows NT 4.0 or later based operating systems.

Building a Control Panel Applet

Building a control panel applet using this toolkit requires you take the following simple steps:

1. Build the control panel framework CPL file using the Desaware Control Panel Applet Wizard.
2. Create a new VB ActiveX DLL project (or modify the template file provided). Perform the modifications listed later in this section.
3. Test the control panel applet by running your VB ActiveX DLL in the VB environment, then installing your CPL file.

Refer to the online documentation for additional information regarding building a Control Panel Applet using the Desaware Control Panel Applet Wizard.

SpyWorks Concepts: Interface Extensions and Hooks

"ActiveX Extensions" is the term that we use to describe four closely related technologies that are implemented by the dwAXExtn.dll component in SpyWorks 5 and later:

- Ability to override the behavior of selected object interfaces.
- Ability to utilize objects through selected interfaces.
- Ability to implement custom and standard interfaces, including those that are not automation compatible.
- Ability to utilize objects that are not automation compatible.

We expect this component to ultimately prove as important to Visual Basic programmers as a subclasser or hook control. It allows you to overcome virtually all of the limitations from which Visual Basic still suffers when compared to Visual C++ in regards to creating ActiveX components or controls.

You will find it considerably easier to understand these features if you already have a background on COM and interfaces and creating ActiveX controls and components. We recommend the book "Dan Appleman's Developing ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed"³ which includes a solid introduction to COM and interfaces designed for Visual Basic programmers (in other words, it does not overwhelm you with implementation details in C++).

Introduction to Interfaces

If you find this section hopelessly confusing, we again encourage you to refer to the book "Dan Appleman's Developing ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed". The following few paragraphs summarize material that is covered through hundreds of pages and dozens of examples in this book.

With Visual Basic 6.0, almost every language element that you deal with is an object. Forms are objects, class modules are objects, controls are objects, and there are a multitude of support objects such as the Printer, App and Collection objects. *All of these objects are based on the Common Object Model, or COM for short.*

When you have a reference to an object (in a variable dimensioned As Object, or as a specific object type), you can't access the data in the object directly. You can only do so by calling functions or accessing properties belonging to the object. These are called the methods of the object.

An object does not expose all of its methods in one group - rather, it divides them into groups called interfaces. Each interface contains methods that are closely related to each other to provide some functionality.

Every time you create a class with functions and properties, you are defining an interface for the class. Interfaces that you create in your own classes are custom interfaces. Windows also defines certain standard interfaces that provide standard functionality, for example: the IViewObject interface is used to display an object. Any object that implements this interface can be displayed in a standard way by any application that uses the object.

³ See <http://www.desaware.com/products/books/com/index.aspx>

You can also conclude from this that it is possible for one object to implement many different interfaces. For example: An ActiveX control is nothing more than a COM object that implements a set of standard interfaces that are required of any ActiveX control (for example: every ActiveX control implements the interface IOleControl). In Visual Basic you can implement multiple interfaces in an object using the Implements statement.

Interfaces are uniquely identified by a 16 byte value called an IID (Interface Identifier). This process is normally hidden to Visual Basic programmers.

There are two standard interfaces that you should keep in mind:

- IUnknown is a standard interface that is a part of every other interface.
- IDispatch is a standard interface that supports late binding - the ability to determine at runtime the set of automation functions that are supported and their parameters.

The IDispatch interface is also called an "automation" interface. Automation interfaces do not support all of the capabilities and variable types that are possible under OLE. Visual Basic does not support all of the capabilities and variable types that are possible even under automation interfaces.

Overriding Interfaces

One of the reasons that Visual Basic is the easiest Windows programming system to use is that it encapsulates much of the functionality of Windows into the language itself. Visual Basic handles many low level tasks for you automatically. However, because of this, you tend to be limited by that very implementation.

For example: the Visual Basic property window displays the value of properties for ActiveX controls that you author using Visual Basic. But what if you wish to override that behavior and display other information? This capability is used by the picture property of a control, where the property window can display the words (bitmap), (icon), or (none) in the property window instead of an image. But Visual Basic does not provide a way for you to customize the display yourself.

The Visual Basic property window display is handled by having each control expose an interface called IPerPropertyBrowsing. This interface also allows a control to create custom drop down lists in the VB property window for any property. But Visual Basic does not allow you to control the behavior of this interface, so you are limited by the implementation built into Visual Basic - which displays the value of a property or a drop down enumerated list based on an enumerated variable.

The dwAxExtn.dll component allows you to override certain methods in selected interfaces such as IPerPropertyBrowsing. The component does not replace the interface - doing so would interfere with Visual Basic's normal operation. It does, however, allow you to intercept certain method calls to add your own functionality as needed.

Referencing Non-Automation Compatible Interfaces

How can you handle cases where you wish to work with an object that exposes an interface that is not automation compatible? There are two issues to consider here:

- The interfaces do not have a public type library. This means that even though the interface is theoretically compatible with Visual Basic, VB does not have a way to determine the methods and parameters of the interface, so it is unable to call them. This may occur because the interface is hidden, or a type library for the interface is unavailable.

- The interface has methods with parameters or properties that are not compatible with automation or one of Visual Basic's data types.

The most reliable way to handle these situations is that used by the `dwAxExtn.dll` component and the `dwEasy3x.ocx` control. These components create separate objects that provide an interface layer to the object with the incompatible interface.

There is another approach that will work in some cases, but at greater risk.

How to Avoid Corrupting Your System Registry

You can use the MIDL compiler included with Visual Basic Professional to create a type library that defines a standard interface.

If the interface is already compatible with Visual Basic, there is no problem with this approach (other than the usual complexity of creating an IDL file), but most compatible standard interfaces are public anyway, so this is not particularly useful.

A second approach is to create and register a type library that changes the definition of a standard interface to use method parameter and property types that are compatible with Visual Basic.

We at Desaware strongly discourage this approach.

Changing the meaning of a standard interface can seriously interfere with the behavior of other applications on your system. All of our products and our own in-house design philosophy are based on minimizing the impact of our tools on the system and providing the highest possible degree of compatibility. We would never permit a developer to redefine a standard interface on any of our systems, and thus do not recommend this approach for others. It's a hack.

If, however, you choose to take this approach, under no circumstances should you ever distribute the new type library with your application. From our perspective this is roughly the equivalent of distributing a virus. It can interfere with the normal behavior of Windows components (and it's hard enough getting them to all work properly together in the first place).

Implementing Standard and Non-Automation Compatible Interfaces

It is one thing to reference standard or non-automation compatible interfaces in other objects. Adding these interfaces to your own objects is another matter entirely. For example: if you wish to distribute ActiveX controls over the Internet, it is important to be able to mark whether the control is safe to load and safe to run. There are two ways to do this: you can mark the control's safety level in the registry during installation, or you can build an interface called `IObjectSafety` into your object. The advantage of this latter approach is that you can create controls that are safe when used in a browser but provide full functionality when used in an environment such as Visual Basic.

The `dwAxExtn.dll` component provides two approaches for implementing interfaces:

- An easy to use implementation of selected standard interfaces such as `IObjectSafety`.
- A general purpose way of implementing any interface that is more flexible, but harder to use.

Declaring and Initializing the dwAxExtn.dll Component

Regardless of whether you are using the component to override an interface, reference a standard interface or implement a new interface, the first steps are the same.

1. Dimension a `dwControlHook` object as follows:

```
Dim ControlHook As dwControlHook
```

You do not need to dimension this `WithEvents`. The `dwControlHook` object does not raise events. You must have one (and only one) `dwControlHook` object in existence throughout the lifetime of the object that you are extending.

2. Create the `dwControlHook` object during the object's **Initialize** event as follows:

```
Set ControlHook = New dwControlHook  
ControlHook.Initialize Me  
ControlHook.EnableComponent(".... license key ...")
```

The first line creates the object. The second line performs the necessary initialization and installs all necessary hooks. How does the component know which hooks to add? That's in the next step. The third line allows you to pass a license key to the component in cases where you wish to distribute it for use with ActiveX controls or other in process components (it is not required for executables). Refer to the section on component licensing for details.

3. Add zero or more `Implements` statements to implement `dwAxExtn.dll` interfaces. For example: to hook the `IPerPropertyBrowsing` interface, you would add the following line at the top of your form, class, `UserDocument` or `UserControl` object:

```
Implements IdwPerPropertyBrowsing
```

4. The `dwAxExtn.dll` component exposes a number of interfaces through which you implement your desired extensions. The component automatically determines which of these interfaces you have implemented during initialization, and enables those features. This approach is both easy to use, and extremely efficient, since it allows all of the functionality provided by this component to be early bound (which offers the best possible performance).
5. Create additional objects from `dwAxExtn.dll` as you need them. Objects might include the `dwEnumerator` object, or the `dwGenericCall` object.
6. Add code for each method of the interfaces that you have implemented. This will be described in the following sections for each interface.
7. Destroy the `dwControlHook` object during the object's **Terminate** event as follows:

```
Set ControlHook = Nothing
```

All of the examples in the on-line Help file assume that you have left the base index of all arrays at zero.

Interface overrides supported in this release of SpyWorks include:

Object	Description
IPerPropertyBrowsing	Controlling the VB Property window.
IOleControl	Trapping Enter keys and detecting changes in container freeze status.

Non-Automation objects exposed in this release of SpyWorks include:

Object	Description
dwEnumerator	Adds For...Each capability to any object (including arrays).
dwGenericCall	Allows you to call methods on any object including those that are not automation compatible.
dwMalloc	Work with objects that expose the IMalloc interface to manipulate blocks of memory.
dwShellLink	Create, edit and view file system shortcuts (See dwEasy3x.ocx).

Standard/Custom interfaces supported in this release of SpyWorks include:

Object	Description
IobjectSafety	Create dual mode controls (safe on the Internet, full featured otherwise).
IdwCustomOleHook	Allows you to implement ANY interface, automation or non- automation compatible, as part of any VB object.

Inside the SpyWorks Interface Extensions

An article by Daniel Appleman

Can one component make it possible to both implement and call any OLE interface even if it is not automation compatible? Find out in this technical white paper.

SpyWorks contains a new component called the Desaware ActiveX Extension Library. This component follows in the tradition of many Desaware products – a very cool technology that provides features that many Visual Basic programmers do not even realize are possible. The purpose of this paper is to introduce some of the ideas behind this technology and how you can use it to do virtually anything in OLE from Visual Basic.

This paper presumes that you are familiar with COM (the Component Object Model) at least to the degree in which it is covered in part I of my recent book: "Developing ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed"⁴. This section introduces COM from the perspective of Visual Basic programmers.

Additional information on objects, interfaces and COM can be found in the on-line reference for the dwAxExtn.dll component.

The COM Contract

A Visual Basic application or component is made up of COM objects. These objects can take many forms. Every class object that you create is a COM object. So are forms and controls. Every time you create an object from a component or system library, you've created a COM object.

Each object has methods available – the functions and properties that you can use to program the object. These methods are grouped together into interfaces. Each interface has a specific set of methods. An object can implement any number of interfaces.

Your application typically accesses these objects through an object variable – a variable that is defined 'As Object', or as a specific object type. Each object variable contains a reference to a single interface on the object.

You can switch between interfaces by assigning one object variable to another type of object variable. For example, the code:

```
Dim ObjectRef1 as MyClass1 ' ObjectRef1 can only reference the MyClass1
interface.
Dim ObjectRef2 as OtherInterface ' ObjectRef2 can only reference an interface
called OtherInterface.
Set ObjectRef1 = New MyClass1 ' MyClass1 is the name of the main interface
for the MyClass1 object.
```

What happens when the following line is executed?

```
Set ObjectRef2 = ObjectRef1
```

⁴ See <http://www.desaware.com/products/books/com/index.aspx>

If the MyClass1 object uses the Implements statement to support the "OtherInterface" interface, this code will work. Otherwise it will fail. You see, every COM interface has a function called QueryInterface that allows one interface to request a reference to another interface. It also has the functions AddRef and Release which are used for reference counting so that objects can be deleted when they are no longer used. These three functions: QueryInterface, AddRef and Release form an interface called IUnknown, which must be supported by every object.

Not only that, but the IUnknown interface is a part of every other interface – thus those three functions define the first three functions of every interface.

There is one more thing that you must keep in mind when dealing with interfaces. While you can and certainly will define your own interfaces, Windows defines many standard interfaces as well – IUnknown is just one of many. Standard interfaces are important because they define ways that clients can work with object without knowing additional details about the object. For example: every object that implements the interface IPersistStream has the ability to save and create itself from a stream – a special kind of storage space that is part of OLE structured storage specification. An application that uses OLE structured storage can store any object that implements this interface – it doesn't need to know details about the object. T

hat's how an application such as Microsoft Word can store data types ranging from sound to video to proprietary graphic objects. It doesn't need to know anything about the object itself. All it needs to do is ask the object for an IPersistStream interface. If it has one, Word can save the object. It's that simple.

The Two Sides of COM

The above description is, of course, very cursory. If you find it unclear, I strongly encourage you to find out more about COM. It should, however, be enough to communicate a key point when it comes to dealing with objects and interfaces.

When it comes to addresses use of objects from Visual Basic, we are actually concerned about two different issues.

First: How do you implement and interface in your object. In other words – how do you make it so that a client using your object can obtain an interface to that object. How could you, for example, implement the IPersistStream interface to your object?

Second: How do you call the methods on an interface once you have a reference to that interface? For example: If your Visual Basic object could implement the IPersistStream interface, how would it call the functions on the stream object itself given a reference to its IStream interface?

Visual Basic provides some support for both of these. You can implement some interfaces using the Implements statement. And you can call methods on many types of objects. But you can't implement every interface, and you can't call the methods on every type of object.

Why not?

The major reason is that Visual Basic does not support all of the data types that are supported by COM. For example: Visual Basic does not have an unsigned long data type. COM demands strong type checking, so Visual Basic simply assumes that it can't implement an interface if it finds a function in it that has an unsigned long parameter. Visual Basic also limits itself to data types that are compatible with OLE automation. OLE automation provides a flexible way for clients to find and call a set of available methods at runtime. This kind of call is handled by a special interface called IDispatch. But functions and properties that can be called through IDispatch can't handle every COM data type. If an interface only uses that subset of

data types that IDispatch supports, it is called OLE automation compatible. Visual Basic can only implement interfaces that are automation compatible.

The same applies to calling methods on an interface. In order for Visual Basic to implement or create an object of a particular interface type, the interface must be publicly defined in a type library, and it must be automation compatible.

Hacking Further

Now, before I go further, I should mention something important about Desaware's philosophy towards extending Visual Basic. We specialize in advanced low level Visual Basic techniques. Many of the techniques that we describe are "dangerous" in the sense that you are very likely to see application crashes and possibly system crashes during the development process. But as a result, we take an extremely paranoid approach both to our components and to the ways that we advocate that they be used. We know that our customers are professionals, and that any approach that we advocate must do more than work correctly. It should also meet the following requirements:

- Follow all ActiveX/OLE, Windows and COM rules of the road.
- The code we recommend should be clear and readable, and supportable on the long term.
- The code and components should not interfere with the operation of other applications or the system.
- The code and components should not interfere with Visual Basic's IDE. You should still be able to break and step through your code. The stop button should not crash Visual Basic.
- The solution should be economical. The cost of the entire package should be far less than it would cost to develop a solution on your own.
- On those rare occasions where we must break one of these rules – we document it and warn people about the consequences of this choice.

Some of the techniques that I'm about to describe can be implemented using pure Visual Basic. Some of the techniques that may be used include:

- Creating a type library that modifies the definition of a system interface to be automation compatible.
- Using the AddressOf operator and memory copying to overwrite internal function table pointers.

Perhaps I'm paranoid, but I simply don't believe in redefining standard system interfaces. You see, it's almost impossible to know when the new definition might interfere with the normal operation of another application. And how can you be sure that the redefined type library will never get off your development system and spread among other systems on your network or to customer sites? In these days of rapid change, it's hard enough getting Windows applications and components to play together reliably anyway. I don't trust myself to remember to undo these types of changes. So I'd rather avoid them in the first place. SpyWorks does not modify any standard interfaces or type libraries.

The SpyWorks code that you'll see does make use of the AddressOf function, and it does perform some function table pointer manipulation internally. But it does so in an extremely paranoid manner, with great care to keep reference counts accurate and to minimize the impact on Visual Basic. You'll find that our interface extension technology in most cases does not interfere with your ability to take full advantage of Visual Basic's development environment. Finally, SpyWorks makes interface extensions easy to do – a matter of minutes in most cases, and as easy as declaring API functions in others.

To demonstrate this approach, let's take a look at a very simple interface called IObjectSafety. This interface is used by controls to report whether they are safe to run in a web browser.

The interface is defined as follows:

```
// Option bit definitions for IObjectSafety:
#define INTERFACESAFE_FOR_UNTRUSTED_CALLER 0x00000001
// Caller of interface may be untrusted
#define INTERFACESAFE_FOR_UNTRUSTED_DATA 0x00000002
// Data passed into interface may be untrusted

// {CB5BDC81-93C1-11cf-8F20-00805F2CD064}
DEFINE_GUID(IID_IObjectSafety, 0xcb5bdc81, 0x93c1, 0x11cf, 0x8f, 0x20, 0x0,
0x80, 0x5f, 0x2c, 0xd0, 0x64);

interface IObjectSafety : public IUnknown
{
public:
    virtual HRESULT __stdcall GetInterfaceSafetyOptions(
        /* [in] */ REFIID riid,
        /* [out] */ DWORD __RPC_FAR *pdwSupportedOptions,
        /* [out] */ DWORD __RPC_FAR *pdwEnabledOptions) = 0;

    virtual HRESULT __stdcall SetInterfaceSafetyOptions(
        /* [in] */ REFIID riid,
        /* [in] */ DWORD dwOptionSetMask,
        /* [in] */ DWORD dwEnabledOptions) = 0;
};
```

The operation of this interface (what the various functions do) won't be covered here – you can find the explanation in my ActiveX book. You don't need to know what they do in order to learn how to handle them in Visual Basic.

This interface suffers from several problems from the Visual Basic perspective. Though a standard interface, most Windows systems do not include a type library for the interface. Even if they did, the interface as defined here uses DWORD data types – an unsigned long type that is not compatible with Visual Basic. The REFIID type is actually a pointer to a 16 byte GUID value, which also is not an automation data type.

Implementing IObjectSafety – The Easy Way

SpyWorks interface extensions are implemented using the dwAxExtn.dll component. First you must add a reference to the Desaware ActiveX Extension Objects Library to your application. You'll use the dwControlHook object to enable the interfaces that you want to implement. Each interface is implemented with the aid of a Desaware defined interface called IdwCustomOleHook. You'll also need an array to hold the GUID values of the interfaces that you're implementing.

The following declarations will typically appear in the module declaration level of the class or control that you are defining:

```
Implements IdwCustomOleHook
Dim Interfaces(15, 1) As Byte
Dim ctlhook As dwControlHook
```

When you initialize the object, you'll do the following:

```
Set ctlhook = New dwControlHook
Call ctlhook.Initialize(Me)
```

This code creates the main `dwControlHook` object and passes it a reference to the object itself using the `Initialize` method. The first thing that this method does is query the object using `QueryInterface` to see which ActiveX extension interfaces you have decided to implement. The `IdwCustomOleHook` interface that is implemented in this example has the ability to allow you to implement any interface in your application. So the object begins by calling methods on the `IdwCustomOleHook` interface to find out about the interfaces that you want to implement.

It calls the `IdwCustomOleHook_GetInterfaceCount` interface to allow you to tell the object how many interfaces you wish to implement. In this case, only one interface (`IOjectSafety`) is implemented as follows:

```
Private Sub IdwCustomOleHook_GetInterfaceCount(iCount As Long)
    iCount = 1
End Sub
```

The `dwControlHook` object needs to know which interfaces you wish to implement. You can specify them by name or by GUID using a standard string format. `IOjectSafety` is rarely registered as a standard interface, but you can see below how you could specify `IOleWindow` just by providing the name of the interface. The `dwControlHook` object will search the registry for the interface identifier. In this example, we use the `IOjectSafety` GUID obtained from the C declaration shown earlier. This method will be called once for each of the interfaces that you defined based on the count you specified earlier.

```
Private Sub IdwCustomOleHook_GetInterfaceName(ByVal InterfaceNumber
As Long, InterfaceName As String)
    If InterfaceNumber = 0 Then
        ' InterfaceName = "IOleWindow"
        InterfaceName = "{cb5bdc81-93c1-11cf-8f20-00805f2cd064}"
    End If
End Sub
```

You need to specify a location to store all of the interfaces. It can be a memory buffer or a byte array. In this example we just use a byte array that is defined to be large enough to hold all of the interfaces. The `VarPtr` operator retrieves the location of the start of that buffer. We decided to store the interfaces with the object in this manner so that you could easily obtain the actual GUID, or set it manually if you prefer to do so.

```
Private Sub IdwCustomOleHook_ReferenceInterfaceIID(IIDPointer As Long)
    IIDPointer = VarPtr(Interfaces(0, 0))
End Sub
```

Now you need to provide the addresses of the functions for each interface. The GetAddress function shown below allows you to obtain the address of a function using the AddressOf operator.

```
Private Function GetAddress(ByVal addr As Long) As Long
    GetAddress = addr
End Function
```

The IdwCustomOleHook_GetInterfaceVtbl requests the functions for each of the functions in the interface. You don't need to provide the standard IUnknown functions – they are handled automatically.

```
Private Sub IdwCustomOleHook_GetInterfaceVtbl(ByVal InterfaceNumber As Long, FunctionAddresses() As Long)
    ReDim FunctionAddresses(1)
    FunctionAddresses(0) = GetAddress(AddressOf GetOptions)
    FunctionAddresses(1) = GetAddress(AddressOf SetOptions)
End Sub
```

The GetOptions function demonstrates a typical implementation function. The first parameter is always the object itself. The IUnknown object type is the lowest level generic object point – since every interface includes IUnknown, this object type is guaranteed to work with any interface. You can then easily set it into a variable that is defined 'As Object' or to your own object type, and call methods on the object directly.

```
Public Function GetOptions(ByVal obj As IUnknown, ByVal riid As Long, X1 As Long, X2 As Long) As Long
    Dim objmine As YourObject
    Set objmine = obj
    ' Perform operation here
End Function
```

This is a key feature to the SpyWorks approach to implementing interfaces. Since you can immediately access the actual object, the amount of code implemented in your global module can be reduced to an absolute minimum. This keeps your program highly modular, making it more readable and supportable in the long term.

That's all there is to it. The dwControlHook object works by hooking the QueryInterface functionality of your object, but it doesn't interfere with the object's normal operation.

What if the interface uses a non OLE automation type? It doesn't matter. You'll use standard Visual Basic data types in the functions that you implement. It's up to you to correctly interpret the meanings of those variables. For example: the riid parameter in this case is a long value that contains a pointer to a 16 byte buffer containing the identifier of the interface that this function is testing. It turns out that for most applications of the IObjectSafety interface, you won't use this parameter at all. But you could easily use a memory copy function to copy the data from the address provided in this variable to a byte array, and thus access the data in Visual Basic. These types of data conversions and manipulations are routine to VB programmers who use the Win32 API. Many of them are discussed in my book *"Dan Appleman's Visual Basic Programmer's Guide to the Win32 API"*.

Implementing IObjectSafety – The Easier Way

As easy as the generic implementation approach is, there are a number of interfaces that we tried to make even easier to handle by building direct support for those interfaces into the dwControlHook object. IObjectSafety is one of these.

Instead of implementing the `IdwCustomOleHook` interface, you can simply interface the `IdwObjectSafety` interface. The `dwControlHook` object will discover that you've implemented this interface when it is initialized and will handle all of the interface manipulation directly.

All you need to do is implement the `IdwObjectSafety_GetInterfaceSafetyOptions` and `IdwObjectSafety_SetInterfaceSafetyOptions` in your object directly. No need to specify GUID's, or create module level functions.

```
Dim ControlHook As dwControlHook
Implements IdwObjectSafety

Set ControlHook = New dwControlHook
' Set up the control hook
Call ControlHook.Initialize(Me)

Private Sub IdwObjectSafety_GetInterfaceSafetyOptions(ByVal piID As Long,
SupportedOptions As Long, EnabledOptions As Long)
' This control is always safe for initialization and safe for scripting
SupportedOptions = DwAXExt.dwSafeToInitialize Or
DwAXExt.dwSafeToProgram
EnabledOptions = SupportedOptions
End Sub

Private Function IdwObjectSafety_SetInterfaceSafetyOptions(ByVal piID As
Long, ByVal OptionMask As Long, ByVal EnabledOptions As Long) As
Boolean
' We're always safe, so just return True
IdwObjectSafety_SetInterfaceSafetyOptions = True
End Function
```

The `dwAxExtn.dll` object supports a number of standard interfaces directly including `IPropertyBrowsing`, `IEnumVariant` and `IOleControl`.

Calling Generic Interfaces

You've seen how an object can implement the `IOBJECTSafETY` interface. What if you have an object reference and you want to obtain an `IOBJECTSafETY` interface for the object and call the methods of that interface?

We spent a great deal of time trying to choose the most reliable way to generically call any method on any interface. The one we chose turned out to be the simplest, and has the advantage of running flawlessly both in the design environment and in compiled applications.

After adding a reference to the `dwAxExtn.dll` object, you must create a `dwGenericCall` object for each object that you wish to call. Let's say you have object "TargetObj" which implements `IOBJECTSafETY`.

```
Dim gencall As dwGenericCall
Dim TargetObj As IUnknown
```

When you're ready to work with the object, you would have the following code:

```
Set TargetObj = Some function that returns an object interface. The interface
need not be IOBJECTSafETY.
Set gencall = New dwGenericCall
```

```

Call gencall.SetInterfaceInfo("{cb5bdc81-93c1-11cf-8f20-00805f2cd064}",
TargetObj)
Set TargetObj = Nothing

```

The SetInterfaceInfo method does two things. First, it lets you specify the interface that you want to call. The format is the same one used earlier in the IdwCustomOleHook interface – you can provide the interface name, or the interface identifier in string format. You also provide it with the object itself. You can delete the object at this point. The dwGenericCall object will hold a reference to the object until you delete it.

The next step is an interesting, but effective trick. The dwAxExtn.dll DLL contains an entry point called dwGenericCall which can be called directly from Visual Basic using standard declaration techniques. You should create a separate alias for each function that you want to call, but they all use the same entry point "dwGenericCall". The first parameter is a special long parameter called ObjectReference – more on this later. The rest of the parameters are the parameters you would use if the method was an API call. You use standard API declaration techniques to define the VB equivalents for each data type. The examples for IObjectSafety are shown below.

```

Declare Function GetOptionsCall Lib "dwaxextn.dll" Alias "dwGenericCall"
(ByVal ObjectReference As Long, ByVal riid As Long, X1 As Long, X2 As
Long) As Long

```

```

Declare Function SetOptionsCall Lib "dwaxextn.dll" Alias "dwGenericCall"
(ByVal ObjectReference As Long, ByVal riid As Long, ByVal X1 As Long,
ByVal X2 As Long) As Long

```

I know it looks strange, but yes – you can specify as many parameters as you need for the function call. The dwAxExtn.dll function fixes things up internally so that as long as you define the declaration correctly for the interface method you are calling, the parameters will all arrive in the right place and the internal stack will clean be cleaned up properly when the call returns!

The call itself is trivial. The only trick is the first parameter. You pass the function the result of the dwGenericCall object's GenericCallReference method, passing the position of the desired method as a parameter. The GetInterfaceSafetyOptions method of the IObjectSafety interface is the third method on the interface (remember, the first three, numbered zero through two, are QueryInterface, AddRef and Release – from the IUnknown interface).

```

Call GetOptionsCall(gencall.GenericCallReference(3), &H5555, 11, 12)
Call SetOptionsCall(gencall.GenericCallReference(4), &H6666, 8, 9)

```

That's all there is to it. The first of these is equivalent to calling the 3rd method of the original TargetObj function with parameters &H5555, 11, and 12. These particular values are, of course, meaningless for this particular interface - which only emphasizes the fact that this is a truly generic calling scheme. The values that you pass and their meanings are entirely your responsibility.

There are many situations where OLE functions and interfaces give you interface references to standard interfaces such as IStream, IMalloc, IStorage and others. The generic call capability shown here allows you to easily work with those interfaces without worrying about side effects.

SpyWorks Concepts: Winsock - Internet/Intranet Programming

There are dozens of controls available to add Internet and Intranet features to your application including several controls that are part of the Visual Basic package. All of these controls suffer from the same limitations of every other control - you are tied down to the features that are offered by the control. Even the most powerful controls rarely expose all of the capability of the underlying Winsock API.

SpyWorks offers a different approach.

The dwSock8.dll component has several unique characteristics:

- It provides a great deal of low level access to the underlying Winsock API.
- It is written in Visual Basic.
- It includes complete source code.

This edition of SpyWorks includes the dwSock component that is designed for version 1.1 of Winsock.

How to Approach the Winsock Package

This component can be as complex or as simple as you need it to be.

What does this mean?

- If all you want to do is read FTP or web sites, you can use the dwFTPclient or dwHTTP10 objects to perform most operations with very little effort.
- If you want to create some typical sockets for use as server or client applications, you'll find it almost as easy using the dwSockets and dwSocket object.

But this component really was designed for advanced users who really want full control over the Winsock subsystem. So things can get complex very quickly. Don't let the documentation here (which is certainly sparse considering the scope of the subject) intimidate you. Your best source of information will be the sample programs that demonstrate how to perform common tasks.

Important Note Regarding Support For This Component

The Winsock API is quite large and fairly complex. The dwSock component provides a fairly complete wrapping of the Winsock API. If you find any errors in any of the methods or properties of this component, please bring them to our attention and we will correct them as quickly as possible.

Note, however, that we assume that people using this component are reasonably knowledgeable with regards to Winsock programming. Desaware cannot at this time provide additional documentation or training on Winsock programming or particular applications or protocols that use this type of programming. Nor can we provide aid with regards to obtaining an Internet connection, configuring networks or servers, or other system administration tasks.

In other words, while we have provided some simple examples that you can use to perform common tasks (such as perform name resolution, retrieve a file via FTP or retrieve a web page), we do expect that you will have to learn the principles of Winsock API programming and TCP/IP from other sources if you wish to perform more sophisticated tasks. Entire books have been written on these subjects, and we simply cannot cover that much material in this documentation.

Learning Winsock

A complete description of sockets and the Winsock API is far beyond the scope of this document. You will need to be familiar with the fundamentals of Internet programming in order to take full advantage of this component.

The Winsock API documentation in the Win32 SDK or MSDN CD-ROM library will prove useful.

The book "Visual Basic 6.0 Internet Programming" by Carl Franklin (ISBN 0-471-31498-6) contains an excellent introduction to Internet programming.

The Microsoft Developer's Network CD-ROM includes the complete Winsock specification. This component is written to version 1.1 of the specification.

Meanwhile, here are a few introductory concepts to help you get started.

IP Addressing

Every system on the Internet has a unique address. This address is called an IP address (which stands for Internet protocol). The address is currently 32 bits long.

When you contact a system, such as www.desaware.com, the name "www.desaware.com" identifies a system, but is not the actual IP address. This kind of identifier is also called a URL or Universal Resource Locator (for reasons that are probably well known to Internet mavens but are totally irrelevant for this discussion). When you request this site, your system uses the name resolution capabilities of the Winsock API to look up the IP address assigned to that name. This lookup might actually involve a search of many systems over the Internet - but that takes place behind the scenes. In this case the Winsock functions would tell you that the IP address of www.desaware.com is 209.35.183.216. These numbers represent four bytes of data which form a 32 bit IP address.

The order of bytes in the 32 bit long address is important. The Internet defines a specific order called the "network order" for the bytes, but your system may pack bytes into a long variable in a different order, depending on the processor you are using. This order is called the host order. Winsock provides functions that allow you to convert host order data into network order data, so you can use your computer's natural ordering until right before you use the address.

Ports

Let's say you want to connect to a system. You have the other system's IP address and you send it a message telling it that you want to make a connection. How does the system know what type of connection you are making? Are you trying to transfer a file using FTP or request a web page? And what if many systems are requesting connections at once - how does the system keep track of the connections?

These problems are solved by having each system support multiple ports. A port is just a number that identifies a connection point to the system. Some port numbers define standard types of connections. For example: if you connect to a system at port 21, you are requesting an FTP connection. Port 80 is a HTTP request (HTTP is the world wide web protocol).

Every connection thus has the following attributes:

- A source IP address (your system's address).
- A source port (the port that you are using for the connection).
- A destination IP address (the address of the system to which you are connecting).
- A destination port (the port that you are connecting to on the destination system).

All four of these attributes form a unique connection.

A connection is made up of two sockets, one on each system. The socket is defined by an IP address and port combination.

When a client system requests an FTP connection it first creates a socket on its own system using an available port number over 1025. It then attempts to connect to port 21 on the server system.

Meanwhile, the server has created a socket that is bound to port 21 using the Bind function, and told to listen using the Listen function. When the server is notified that a client has requested a connection, the server creates a new socket at an available port number and connects that socket to the client's socket. This leaves the original server socket available to listen for further connections.

UDP and TCP

There are two protocols that are commonly used by socket connections (though others are supported). UDP is a connectionless protocol. This means that when you send data across a UDP socket, you can't be certain that the data will actually arrive, and no error will be generated if it does not. TCP sockets form reliable connections, meaning that data is guaranteed to arrive at the destination - if it does not, an error will be reported.

Most of your internet programming will probably use TCP (the FTP and HTTP protocols both use it), but UDP does have its uses. It is much more efficient, for example, and is ideal for situations where lost data is not a big issue, such as some audio data streams.

dwSock Architecture

The most important objects in the dwSock component are the dwSockets object and the dwSocket object.

The dwSockets object initializes the Winsock system and keeps track of all of the sockets that you have in use. You will usually use only one dwSockets object, though higher level objects may create and use their own dwSockets objects as well to manage their own set of sockets. If you are using the dwSock package from a multithreaded client (such as an ActiveX control marked for apartment model threading) you will create a separate dwSockets object in each apartment. This component is not designed for free threading.

The dwSocket object represents an individual socket. Its methods correspond to Winsock API commands that control individual sockets (and take socket handles as parameters). This is the object that you will actually use to perform data transfers.

The Desaware Winsock Component is designed to take full advantage of the asynchronous features of the Winsock DLL. This means that most requests simply start a background operation. This is the only way to provide reasonable performance with multiple connections - otherwise a data request could block all other sockets in use by the application. The dwSockets object raises events when background operations finish, passing the appropriate object to the client as an event parameter. All of the work of keeping track of which operation refers to which socket, is handled by the component.

The dwSocketUtil object includes a number of utility methods that simply wrap some useful Winsock API functions. It allows you to request various types of information from the internal Winsock database. For example: you can obtain information about a service by requesting a dwServEnt object. Information about protocols can be determined by requesting a dwProtEnt object.

The dwHostEnt object identifies a computer, both by URL and by IP address, and is generally obtained by one of the name resolution functions in dwSockets or dwSocketUtil.

The dwAsyncSocket object manages all of the background operations for sockets. You will probably never use this object directly, as it is designed to be used by the dwSockets object.

The dwUDP and dwTCP objects provide a higher level interface to the Winsock package. You will typically use them instead of a dwSockets object.

The dwFTPclient and dwHTTP10 objects provide easy to use FTP client and HTTP data transfers. These are likely to be the objects that you will use most often in real operations.

Refer to the Winsock specification for a complete list of Winsock errors. The constants for these errors start with the prefix WSA and can be found in the GlobalInfo standard module. The dwsock8.dll object can also raise the following errors:

ERR_INVALIDPARAMETER = vbObjectError + 513	An invalid parameter was passed to a method.
ERR_CMDSTATEERROR = vbObjectError + 514	You requested an operation that is not valid for the current object state.
ERR_HOSTFILEERROR = vbObjectError + 515	A host file error occurred.
ERR_INVALIDHTTPURL = vbObjectError + 516	An invalid URL format was passed to a URL parameter.

Dependencies:

The dwsock8.dll component requires that you distribute the following additional components:

dwshvb8.dll (dwSock6.dll requires dwshvb86.dll instead)	The Desaware Windows Utility component.
dwspy5.dll	A utility DLL used by dwshvb8.dll.
SockIntf.dll	A DLL that is required as an interface to Winsock in

order to retrieve accurate error status information.

SMTP

The dwSMTP8 component is an extension component to the dwSock component that implements the Simple Mail Transport Protocol (SMTP). This is the standard mail protocol that is used by virtually all Internet TCP based Email systems.

Like the dwSock component, the dwSMTP8 component provides a great deal of flexibility. It is easy to use in the form provided, however requires a high familiarity with the SMTP protocol if you wish to go beyond the functionality provided. Full source code for the component is also included.

Using the Winsock Package

Additional examples are included with this package including a simple FTP client and a demonstration of direct use of sockets. Full documentation on the WinSock class objects are found in the SpyWorks Help file.

Winsock Utility Functions

Obtaining Winsock Version Information

Create a dwSockets object, then use it's **SocketData** property to access the dwSocketInfo object which contains information about the current Winsock system. From SktTst1.vbp

```
Dim skt As New dwSockets
Debug.Print "Current version: " & Hex$(skt.SocketData.CurrentVersion)
```

Obtaining Your Host Name

The following code demonstrates how to retrieve your computer's name using Winsock. The dwSocketUtil object provides a number of utility functions.

```
Dim skt As New dwSockets
Dim su As dwSocketUtil
Set su = skt.Util
Debug.Print su.gethostname()
```

Determine the Standard Port Number of a Service

First create a dwSockets object. Next use it's dwSocketUtil object (via the **Util** property) to perform a getservbyname operation, in this case on the "FTP" service. If the operation succeeds, it will display the port number, which in this case is 21.

```
Dim s As dwServEnt, x As Long
Dim su As New dwSockets
Set s = su.Util.getservbyname("ftp", "")
If Not (s Is Nothing) Then
    Debug.Print "Name: " & s.Name
    For x = 1 To s.AliasCount
        Debug.Print.Alias(x)
    Next x
    Debug.Print "Port: " & s.Port
    Debug.Print "Protocol: " & s.Protocol
Else
    Debug.Print "No info on: " & TextServ.Text
End If
```

Perform an Asynchronous Name Resolution

This example from the SockTst2 application demonstrates how names can be resolved.

At the module level define:

```
Dim WithEvents skt As dwsockets
```

Start the operation thus:

```
Dim hndasync&  
hndasync = skt.WSAAsyncGetHostByName("www.desaware.com")
```

One of the following two events will be raised:

```
Private Sub skt_AsyncError(ByVal AsyncHandle As Long, ByVal Errval As  
    Long)  
    Debug.Print "Async Error: " & AsyncHandle & " #: " & Errval  
End Sub
```

```
Private Sub skt_HostResolved(ByVal AsyncHandle As Long, HostObject As  
    dwSock.dwHostEnt)  
    Dim x&  
    Debug.Print "Async Done: " & AsyncHandle  
    Debug.Print "Name: " & HostObject.Name  
    For x = 1 To su.AliasCount  
        Debug.Print su.Alias(x)  
    Next x  
    For x = 1 To su.AddressCount  
        Debug.Print skt.Util.inet_ntoa(su.Address(x))  
    Next x  
End Sub
```

Note how the dwSocketUtil object's inet_ntoa function is used to convert the network address to a string. For a more interesting example, see what happens when you perform a name resolution on www.microsoft.com.

HTTP Example:

The material in this document can be a bit overwhelming. This example illustrates how easy it actually is to retrieve a file from a web site.

At the module level define:

```
Dim WithEvents http As dwHTTP10
```

When the form loads, initialize the dwHTTP10 object

```
Private Sub Form_Load()  
    Set http = New dwHTTP10  
    http.RetrieveMode = dwRetrieveText  
End Sub
```

Here's the command that actually does the request

```
Private Sub cmdExecute_Click()  
    Call http.Execute("www.desaware.com", "GET", , , )  
End Sub
```

The data arrives as a string because we set the **RetrieveMode** property to text

```
Private Sub http_HTTPDataReceived(DataReceived As Variant)  
    Debug.Print DataReceived  
End Sub
```

SpyWorks Concepts: Components and Class Libraries

SpyWorks includes a selection of additional components and classes. A brief description follows. The on-line Help file includes detailed information.

Desaware Windows Utilities and Subclassers (dwshvb8.dll)	A powerful low level component authored in Visual Basic with source code included.
Desaware API Class Library	A set of classes that simplify use of the Win32 API and demonstrate many API programming techniques.
Desaware DLL Background Thread	Create background threads easily with this new component. Run your objects in a separate background thread. (<i>Professional Edition only</i>).
SpyNotes	Additional samples and utilities.

Desaware Windows Utilities and Subclasser

The intent of SpyWorks is to help Visual Basic programmers to do anything in Visual Basic that can be done using other languages. Historically, this meant providing the best low level windows, subclassing and hook tools available. With the appearance of the AddressOf function in Visual Basic 5.0, it became possible for the first time to implement some of these functions in Visual Basic. Desaware's philosophy in cases such as these is: if you can do it in Visual Basic, it's our job to show you how to do it right. The dwshvb8.dll component is written in Visual Basic (though it does make use of a small support DLL dwspy5.dll) and includes full Visual Basic source code. We encourage you to use the actual component in your projects - it is quite efficient and has a great deal of functionality.

Future SpyWorks releases will include advanced technical notes explaining in more detail how the components work internally.

Before you try to use this component or create your own subclasser, please read the section titled Before You Begin in the on-line Help file.

Before you ship this component, please read about Distribution and Licensing Issues.

The following classes are included in the dwshvb8.dll component:

Classes	Description
dwSubClass	A subclassing object.
dwGenericHook	A Windows hook object.
dwPretranslate	A Windows hook object specialized for pre-translation of tab and arrow keys.
dwPrivateWindow	An object that facilitates the creation and use of private windows.
dwScrollBars	Enables window scrollbars

dwObjectList	Provides efficient object collection
dwHandleCol	Maps objects to handles
dwFlexPicture	Converts bitmaps to pictures

SpyNotes

SpyNotes are a series of application notes that will deal with a particular subject and include both code and documentation in Windows Help file format. SpyNotes #1 and SpyNotes #2 (the Common Dialog Toolkit) are included in the Professional edition of SpyWorks. Refer to the on-line Help file for the individual products for further information.

SpyWorks Concepts: Background Threads

Desaware's dwBackThread component offers a mechanism for Visual Basic 6.0 programmers creating ActiveX DLL's to create threads for execution of background operations. The dwBackThread component adheres to all of Visual Basic's threading requirements and all COM threading rules – thus is extremely safe to use. Objects created in background threads are self-synchronizing, so there is no need to worry about explicit synchronization.

The following section summarizes the use of the dwBackThread component, and its object's properties. Complete documentation for the dwBackThread component can be found in the SpyWorks online help file.

Note: We strongly recommend that you read the full online documentation for this component before you attempt to use it. Multithreading of this type imposes strict requirements for software design, debugging and cleanup that are not always intuitive.

dwBackThread - Quick Start

Using the dwBackThread component is easy – just follow these simple instructions:

Using the Project-References menu, add a reference to the Desaware DLL background thread component.

Create a dwObjLaunch object for each background thread you wish to use.

Create a class to run in the background thread. The background class should have a subroutine named ExecuteBackground that takes no parameters.

Declare a variable to hold a reference to the class (you can declare it With Events if you wish).

Create an instance of the background class object using the LaunchObject method of the dwObjLaunch object.

Set the properties for the background class object. Then use the dwObjLaunch BackgroundExecute or BackgroundExecuteDelayed methods to launch the background operation asynchronously.

Let's take a closer look

```
' This line creates a new dwObjLaunch object
Dim BackObjControl As New dwObjLaunch

' This variable will reference the class that runs in the
' background thread
Dim WithEvents BackObj As clsBackTest

' The LaunchObject method here creates an instance of the
' background class object. The background class must be a
' public multi-use class, and must be declared by name.
Public Sub Start()
    Set BackObj = BackObjControl.LaunchObject("BackTest.clsBackTest")
End Sub

' The BackgroundExecute method causes the
' "ExecuteBackground" method in the background class to be
' called asynchronously. You can set properties on the
' background class before calling the BackgroundExecute
' method to set up the operation.
Public Sub BackExecute()
```

```

        BackObjControl.BackgroundExecute
End Sub

' The background class can raise events to let the program
' know when the background operation is finished.
Private Sub BackObj_GotExecute(ByVal ThreadID As Long)
End Sub

' Cleanup is very important!
Private Sub Class_Terminate()
    Set BackObj = Nothing
End Sub

```

Here's a simple background class that simply raises an event when the ExecuteBackground method is called.

Option Explicit

```

Event GotExecute(ByVal ThreadID As Long)

Public Sub ExecuteBackground()
    RaiseEvent GotExecute(GetCurrentThreadId())
End Sub

```

dwBackThread - Methods and Properties

The dwObjLaunch object has the following methods and properties.

LaunchObject(ObjectName As String) As Object

ObjectName is the full name of the object in the form “component.class”. This method creates the background thread and creates the specified object in that thread, returning a proxy to the object that can be referenced from the calling thread.

If you have already used this method to launch an object in a thread, calling this method again will release that object and create a new one. Be sure to free the references in your program held by that object before using this method to launch a new one!

BackgroundExecute()

This method causes the background object's “ExecuteBackground” method to be called. An “invalid method or property name” error will be raised if the background object does not have an ExecuteBackground method, or does not support IDispatch (all VB objects support IDispatch). An error will also be raised if a background execution is already in progress.

The background object's ExecuteBackground method will be called at some arbitrary time after this method is called depending upon the manner in which the system schedules threads. You cannot assume that the background operation will have already started when this method occurs – in fact, it usually will **not** have started.

BackgroundExecuteDelayed()

If no background operation is in progress, this method is identical to BackgroundExecute. If a background operation is in progress, this method signals that the ExecuteBackground method should be called again as soon as it returns. This method is typically used in events raised by the background object, where you wish to start another operation, but a call to BackgroundExecute would raise an error.

BackgroundObject

The dwObjLaunch object holds on to a reference to the background object you create. You can thus access the object at any time using this property. Using this property is safer than holding your own reference because you don't have to worry about releasing the object first if you rely on this property.

dwBackThread - Summary of Rules

A full explanation of these rules and the underlying reasons for them can be found in the online documentation.

- If you call the dwObjLaunch object's BackgroundExecute method while a background execution is in progress, you will get a runtime error. Use BackgroundExecuteDelayed method to queue another call to the background object's ExecuteBackground method while a background execution is in progress.
- Always free background objects (set all references to Nothing) before you free references to the dwObjLaunch object. Failing to do so may trigger a runtime error.
- Do not rely on Visual Basic to clean up your objects – do so during the class terminate event. You don't know what order VB will use to release objects.
- Never free the dwObjLaunch object from within an event that was raised from your background objects. Doing so might cause a memory exception.
- If you use progress events (especially to allow access to your background object) be sure to actually implement the event. If you don't add code for the event, it will be ignored and method calls will not be allowed into the background object at that time.
- If your background object creates additional objects and passes them back to the client, be VERY sure to release those objects as well before you free the dwObjLaunch object.
- If at all possible, Terminate background operations before your client application terminates.
- Watch for automation timeouts that can occur if method or property calls to a background object are blocked waiting for a background operation to end, raise an event, or call DoEvents.
- If you're careless and call the ExecuteBackground method of the background object directly, instead of through the dwObjLaunch object, your application will probably freeze (you'll get OLE automation busy messages, and such). Always call the ExecuteBackground method through the dwObjLaunch object's BackgroundExecute method.
- Be sure to test your background object as a compiled DLL – it will work, but will not exhibit multi-threading when tested within the VB development environment.

- Each `dwObjLaunch` object creates a single object on a thread that it manages. You can, however, create as many `dwObjLaunch` objects as you wish, subject to limitations on the number of threads in the system.
- If you pass object references to your background object, be sure to pass them with `ByVal`. Marshalling `ByRef` objects across threads in VB seems iffy (to say the least).

SpyWorks Concepts: Tools and Utilities

SpyWorks includes a large selection of additional tools and utilities written in Visual Basic. This section includes a brief review of these parts of the package. Detailed instructions for using these tools can be found in the on-line Help file.

Tools	Description
SpyMsg	A program for spying on messages going to a form, control or process. Demonstrates advanced subclassing and hook techniques. Professional Edition includes source code (32 bit VB 4/5/6 only).
SpyWin	A program for browsing the windows of an application or the entire system. Allows you to obtain detailed information about each window. Includes source code, allowing you to learn advanced API and callback techniques.
SpyMem	A program for examining processes and memory under Windows 95/98/ME and Windows NT/2000/XP. Professional Edition includes VB 4/5/6 source code, allowing you to learn about otherwise poorly documented system API techniques.
SpyMenu	A program for examining the menus of any window. Includes source code for VB 4/5/6.

SpyMsg

This advanced utility includes the following features:

- View any or all of the messages going to a form, control or process.
- Your choice of detection technologies to use: subclassing, SendMessage hooks or GetMessage hooks.
- Set the scope of message detection: Any combination of one or more windows, any process, or system wide.
- View detailed information on the form or control that received the message.
- View information on the parameters to the message
- Easy selection of Windows to subclass - use a Windows hierarchy "tree" or point at a window and click.
- Write output to a file to create a complete history of the messages that occur in your application.
- Improved parameter descriptions for most standard windows messages.
- Full VB6 Source Code.

SpyWin

This advanced utility includes the following features:

- The first truly Visual Basic aware window/form/control browser (16 bit only).

- Windows are organized by task or process, making it easy to track down the windows for a given application.
- Provides a great deal of information about each window/form or control such as control name and model information, styles, class information and so on.
- Allows you to select windows for further processing by your own programs.
- SpyWin includes complete VB and .NET source code.

SpyMem for Windows

This advanced utility includes the following features:

For Windows 95/98/ME

Browse the Windows process, stream and module lists. Free any module or stop any process (ideal for cleaning up after a GPF).

Browse the Windows memory heap of each process. View any block of memory.

Automatic comparison of the current heap with a saved reference - makes it easy to track down memory that is not being freed.

Ability to measure the memory used by an application when it is loaded.

For Windows NT/2000/XP

Browse the Windows process lists. Terminate any process.

Browse the Windows memory heap of each process.

Automatic comparison of the current heap for a process with a saved reference - makes it easy to track down memory that is not being freed.

Ability to browse the global driver list..

Includes complete VB 6 source code.

SpyMenu

This utility includes the following features:

- Analyze the system menu and regular menu for any window in the system.
- Includes complete VB 6 source code.

SpyWorks Concepts: Callbacks

Consider a situation where you need to enumerate objects. For example: the problem of listing all of the top-level windows in the system. One way to do this is to use the EnumWindows API function. This function works by calling a user-defined function for each window. Similar enumeration functions follow the same principle for enumerating fonts, properties, GDI objects, and so on.

How do enumeration functions know which user-defined function to call? Like data objects, the code for functions are present in memory and has a memory address associated with it. The enumeration functions require as one of their parameters the address of a user-defined function to call. Figure 6 illustrates the program flow used during such enumeration. The Windows application passes the address of a callback function to Windows. Windows then calls the function for each object being enumerated.

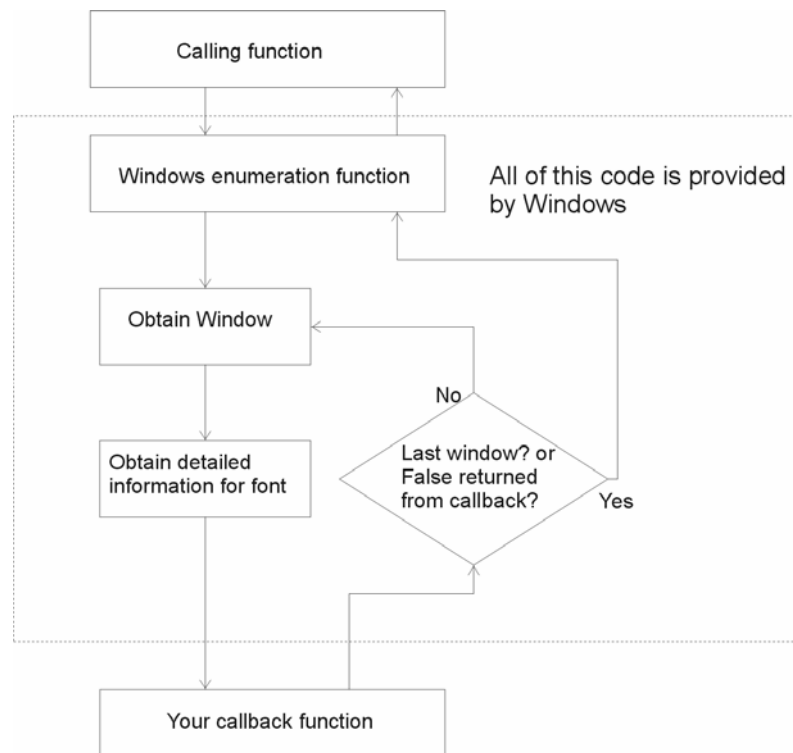


Figure 7 - Illustration of Callbacks

Visual Basic 5.0 and later allows you to obtain an address of a public VB function in a standard module to use as an enumeration function. For earlier versions of Visual Basic, the dwcbk32.ocx custom control contains a built-in pool of function addresses that can be provided to an enumeration function.

The dwcbk32.ocx control is available in SpyWorks 7.1. Refer to the SpyWorks 7.1 documentation for further details. VB6 programmers would only need this component if they needed to be able to handle callbacks using non-standard callback conventions.

Distribution and Licensing

Desaware's philosophy is that you should be able to distribute any components and applications that you create using our tools without having to pay any distribution royalty fees.

The following topics discuss the files that are re-distributable, and special licensing requirements that exist when distributing the dwAxExtn.dll, dwbkthrd.dll, dwsock8.dll, dwsmt8.dll, dwcmndg6.dll and dwshvb8.dll components, and the dwsbc80.ocx, dwshk80.ocx and dweasy80.ocx controls when used to build your own in-process components.

About our Software License

With the increased ability of Visual Basic to allow you to create components, we have adapted our software license to be comparable to those of Microsoft and other vendors with regards to building components based on our components.

Desaware's software components and controls are provided with the intent that they be used in applications and components that offer significant and primary functionality beyond that of our components. In other words - our license does not permit you to use code or software that we provide to market another component that provides the same or slightly modified functionality that we provide. For example, you can't use dwshvb8.dll to create another subclassing or hook tool that you will market.

You may not distribute the source code provided with SpyWorks to anyone who has not purchased SpyWorks version 8 or later.

You may not, under any circumstances, distribute the file dwcmndlg.lic or dwsdes32.dll.

You may, however, distribute our components on a royalty free basis with your compiled applications. You may distribute modified versions of our component for which we provide source code with your compiled applications as long as you include either our licensing code or other code that will prevent the component from being programmed without a valid SpyWorks license. You must also change the project name and rebuild the component with compatibility turned off in order to generate a completely new set of class and object identifiers. If you have any questions regarding distribution, please do not hesitate to contact us directly.

Licensing dwAxExtn.dll, dwbkthrd.dll, dwshvb8.dll, dwcmndg6.dll, and dwsock6.dll

The dwbkthrd.dll, dwshvb8.dll, dwsock8.dll, dwcmndg6.dll and dwaxextn.dll components, like other Desaware components, have built in licensing that require that a license file be present in order for the component to run within the Visual Basic environment. This poses no problem when using the component with applications or ActiveX EXE servers. However, it does present a problem if you wish to use these components within your own ActiveX DLLs, ActiveX controls or ActiveX Document DLLs. When the end-user tries to load your component within Visual Basic, even though your control may be licensed, the dwbkthrd.dll, dwshvb8.dll, dwsock8.dll, dwcmndg6.dll and dwAxExtn.dll components detect that they are running within Visual Basic and raise a licensing error.

In order to allow you to use these components within your own components, Desaware is introducing a license key scheme that will allow you to enable the component to run even when a license file is not present. This scheme requires only a few lines of code in your component. Naturally, there are no costs or royalty fees

for you to use this mechanism. However, we do remind you that under the terms of the SpyWorks 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 subclassing, hook, winsock, common dialog, ActiveX extension or window management 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 re-distributable components to implement your own general purpose database product. If you have any questions, don't hesitate to contact us.

Using the License Key with dwshvb8.dll

The dwshvb8.dll project includes a class called "dwRedistributable". This class contains a single method called "IsLicensed". When you call this function with a valid license key, the dwshvb8 component will be enabled for use with your component. You must create a dwRedistributable object and call the IsLicensed method before you attempt to create any other objects in the dwshvb8 library.

Here is an example of how the component can be enabled for redistribution. This sample enables the dwshvb8.dll component to run in VB when the dwsock8.dll component is also present.

```
Dim sublicok As Boolean
Dim EnableRedistribution As New dwshvb8.dwRedistributable
' Enable the SpyWorks component when used with this component
(dwsock8.dll)
sublicok = EnableRedistribution.IsLicensed("3636242E302E6A213E2D57")
```

The IsLicensed method will return True (non-zero) if the component licensing is enabled for the component. Note that this key will only work to enable licensing when using your compiled component. You'll need the SpyWorks license file while designing your control.

Using the License Key with dwsock8.dll

The dwsock8.dll project includes a class called "dwRedistributable". This class contains a single method called "IsLicensed". When you call this function with a valid license key, the dwsock component will be enabled for use with your component. You must create a dwRedistributable object and call the IsLicensed method before you attempt to create any other objects in the dwsock library.

Here is an example of how the component can be enabled for redistribution for an imaginary control named desaware.ocx:

```
Dim sublicok As Boolean
Dim EnableRedistribution As New dwSock.dwRedistributable
' Enable the SpyWorks component when used with this control (desaware.ocx)
sublicok = EnableRedistribution.IsLicensed("36242420242436207C2E343953")
```

The IsLicensed method will return True (non-zero) if the component licensing is enabled for the component. Note that this key will only work to enable licensing when using your compiled component. You'll need the SpyWorks license file while designing your control.

Using the License Key with dwAxExtn.dll

The latest version of the dwAxExt component (now named dwaxextn.dll) supports licensing through the new EnableComponent method of the dwControlHook object. This method takes the license key as a parameter.

The following code shows how you might enable licensing for an imaginary Visual Basic ActiveX control named `desaware.ocx` that uses the `dwAxExtn.dll` component.

```
Dim ctl As New dwControlHook

Private Sub UserControl_Initialize()
    On Error GoTo initfailed
    ctl.EnableComponent "36242420242436207C2E343953"
    ctl.Initialize Me
Exit Sub
initfailed:
    Do appropriate error handling here.
End Sub
```

A licensing error #419 (Permission to use object denied) if you attempt to use the `dwAxExtn.dll` component within the VB environment without enabling licensing.

Using the License Key with `dwbkthrd.dll`

The `dwbkthrd` component supports licensing through the `EnableComponent` method. This method takes the license key as a parameter.

The following code shows how you might enable licensing for an imaginary Visual Basic ActiveX DLL file named `BackTutorialDLL2.dll` that uses the `dwbkthrd.dll` component.

```
Dim backlauncher As dwObjLaunch

Set backlauncher = New dwObjLaunch
' License the background thread component
' to run with BackTutorialDLL2.dll
backlauncher.EnableComponent _
    "1020342A0730302A2028362D170908777C253B2D53"
Set backclass = _
backlauncher.LaunchObject("BackTutorialDLL2.BackClass2")
```

A licensing error #419 (Permission to use object denied) if you attempt to use the `dwbkthrd.dll` component within the VB environment without enabling licensing.

Creating a License Key

You can create a license key for your component 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 past into your application.

Final Notes on Licensing DLLs

- You only need to use this license key scheme when distributing in-process components such as ActiveX DLL's, ActiveX Controls or ActiveX document DLL servers. You do NOT need to use this mechanism for standalone applications, ActiveX executables, and ActiveX document EXE servers. They do not run within the Visual Basic environment, thus need no special handling.

- If you release the dwshvb8 objects that you are using such that the component unloads, you will need to re-enable the component using the license key once the component is reloaded. It is perfectly safe to call the IsLicensed function multiple times. It is also safe to maintain a reference to this object to prevent the dwshvb8 component from unloading until your component terminates.

Licensing dwsbc80.ocx, dwshk80.ocx and dweasy80.ocx for use as Constituent Controls

The new dwsbc80.ocx, dwshk80.ocx and dweasy80.ocx may be used as constituent controls in ActiveX controls created with Visual Basic. As before, our licensing terms require that your control provide significant and primary functionality beyond these controls if you are using them as constituent controls.

Because of the mechanism that Visual Basic uses to create constituent controls, it is not possible to use the object based licensing scheme that we use for our ActiveX DLLs. Instead, it is necessary for you to create a license key and place it in the registry.

Creating a license key is described in the section titled "Licensing dwAxExtn.dll, dwbkthrd.dll, dwshvb8.dll, dwcmndg6.dll and dwsock8.dll". Use the compiled name of your OCX to generate the license key.

When you install your control, add this license key into the registry as a subkey under location

HKEY_CLASSES_ROOT\Licenses\DesawareConstituentOCX

Thus, to enable constituent control operation for an ActiveX control named test.ocx, you would create the following key:

HKEY_CLASSES_ROOT\Licenses\DesawareConstituentOCX\262424357D2A273D52

Most installation programs (including downloaded CAB files) provide for the addition of registry entries. Refer to your installation program documentation for more information.

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 in the license agreement.

File	Description
DWSHENGINE80.DLL	Main runtime dynamic link library. This file is used by the extension controls dwsbc80.ocx, dwcbk.ocx, dwshk80.ocx and dweasy80.ocx. It should be installed in a directory that is in your PATH environment setting (typically the system directory).
DWSHUTL80.DLL	Utility dynamic link library used by dwshvb8.dll.
DWSBC80.OCX	Generic Subclass custom control. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This

	file can be distributed with your compiled Visual Basic application without any license fees. (Note that dwshengine80.dll must be present to use this control.
DWSHK80.OCX	Keyboard and Windows Hook custom control. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This file can be distributed with your compiled Visual Basic application without any license fees. (Note that dwspydll.dll or dwshengine80.dll must be present to use this control.
DWEASY80.OCX	Easy subclassing custom control. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This file can be distributed with your compiled Visual Basic application without any license fees. (Note that dwshengine80.dll must be present to use this control.

DWSHVB8.DLL	SpyWorks VB 6.0 Subclasser component. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This file can be distributed with your compiled Visual Basic application without any license fees, however it does require that you take additional steps to enable the component when distributed with your ActiveX controls and in-process components. (Note that dwshutl80.dll must be present to use this component.)
DWAXEXTN.DLL	SpyWorks VB ActiveX extension component. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This file can be distributed with your compiled Visual Basic application without any license fees, however it does require that you take additional steps to enable the component when distributed with your ActiveX controls and in-process components.
DWSOCK8.DLL	Winsock ActiveX component. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This file can be distributed with your compiled Visual Basic application without any license fees, however it does require that you take additional steps to enable the component when distributed with your ActiveX controls and in-process components. (Note that dwshutl80.dll, dwshvb8.dll, and sockintf.dll must be present to use this control.)
DWSMTP8.DLL	SMTP ActiveX component. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This file can be distributed with your compiled Visual Basic application without any license fees. (Note that dwshutl80.dll, dwshvb8.dll, dwsock8.dll and sockintf.dll must be present to use this control.)
DWBKTHRD.DLL	Background thread ActiveX component. This file should be installed and registered in a directory that is in your PATH environment setting (typically the system directory). This file can be distributed with your compiled Visual Basic application without any license fees.
Export Alias DLLs	Export Alias DLL's created using the

dwExUtil.exe or ExportWizard.exe programs may be distributed.

The following files and controls may NOT be distributed and are subject to the terms in your license agreement.

File	Descriptions
dwsdes32.dll	Design time dynamic link library. This file must be installed in the same directory as dwshengine80.dll.
Projects from the “\SpyWorks\VB ? Samples\” and “\SpyWorks\VS_NE T_Samples\” and associated files	The sample source files for SpyWorks. Use these as examples of possible applications for SpyWorks.
spydemo.vbp and associated files	The source files for the SpyDemo application. Refer to the file spydemo.vbp for a complete list of the files in this project. Use these as examples of possible applications for SpyWorks.
spymsg32.exe, spymsg.vbp and associated files	SpyWorks Windows Message Monitor. Refer to the file spymsg.vbp for a complete list of the files in this project.
spywin32.exe, spywin.vbp and associated files	SpyWorks Windows Hierarchy Browser. Refer to the file spywin.vbp for a complete list of the files in this project.
spymnu32.exe, spymenu.vbp and associated files	SpyWorks Menu Hierarchy Browser. Refer to the file spymenu.vbp for a complete list of the files in this project.
spymem32.exe, spymem6, spymem??.vbp and associated files	SpyWorks Memory Browser. Refer to the file spymem.vbp for a complete list of the files in this project.
swiniedt.exe, swiniedt.mak and associated files	SpyWorks.ini Initialization File Editor. Refer to the file swiniedt.vpb for a complete list of the files in this project.
Dwspywrk.hlp	On-line Help file for all SpyWorks controls and applications.

spynote1.hlp and sn1*.vbp and associated files	Application Note Pak #1 for SpyWorks.
spynote2.hlp, project files in the	Application Note Pak #2 for SpyWorks. Demonstrates how to modify the Windows

SpyNote2 subdirectory	Common Dialogs.
dwshvb8.vbp and associated files	The source files for the dwshvb8 ActiveX components. Refer to the dwshvb8*.vbp files for a complete list of the files in this project.
dwtaskbr.vbp and associated files	The source files for the dwTaskbr and dwTaskb6 ActiveX controls. Refer to the Taskbar*.vbp files for a complete list of the files in this project.
dwcmmndlg.vbp and associated files	The source files for the dwcmmnDlg.DLL component. Refer to the file CommnDlg.vbp for a complete list of the files in this project.
dwsock.vbp and associated files	The source files for the dwsock8.dll component. Refer to the file dwsock.vbp for a complete list of the files in this project.
dwsmtip.vbp and associated files	The source files for the dwsmtip.DLL component. Refer to the file dwsmtip.vbp for a complete list of the files in this project.
dwExUtil.exe	SpyWorks Dynamic Export Utility program. This utility creates the alias export DLL.
dwExport.dll, dwExport.tlb	SpyWorks Dynamic Export DLL. This DLL is used to create the alias export DLL.
CPLWizard.exe, dwcpl.svt, cpapplet.tlb, Svcchelp.dll	SpyWorks Control Panel Applet Wizard and dependency files.
ExportWizard.exe, dwNetExp.xft, Sw7help.dll	SpyWorks .NET Function Export Wizard and dependency files.

SpyWorks and Visual Studio .NET

The .NET framework represents a completely new “virtual machine” from the perspective of both Visual Basic and C++ programmers. SpyWorks has historically been a product dedicated to providing high level access to lower level system functionality. With the arrival of .NET, some of the previous features of SpyWorks are now handled by the .NET framework, while others are even more important. Our focus with SpyWorks has been to ensure that key SpyWorks capabilities will be available for .NET in as timely a manner as possible.

Refer to the SpyWorks 8.0 Manual for Visual Studio .NET for detailed information on using SpyWorks with .NET, including use of the native .NET Subclass and Hook components, and migration of projects from VB6 to .NET

Additional Topics

Shmessages.ini - Configuration and Initialization File

The SHmessages.ini initialization file can be used to configure the interpretation of message and class names under SpyWorks. Please refer to the on-line Help file for further information.

This divides into a number of functional areas:

Message Grouping	How messages are divided into groups.
Style Interpretation	How SpyWin knows which styles to use for a control.
Message Interpretation	How SpyMsg knows which message name to assign to a message value.
Application Specific	Where the SpyWorks applications save their current settings.

Each SpyWorks ActiveX custom control loads most of the SHmessages.ini information when it is loaded. As a result, if you change the contents of this file it is necessary to close all applications using the control for the changes to take effect for that particular control.

In order to save time, a precompiled version of SHmessages.ini is saved to disk in file SpyWorks.sts. If no changes have occurred between uses of SpyWorks, the message information will be loaded from this status file instead of SHmessages.ini.

The dwsbc80.ocx custom controls use the information in SHmessages.ini at design time when you are specifying message filters. This information is not used at runtime.

The SHmessages.ini file is not required for executable applications that use the SpyWorks controls and thus should not be distributed with your compiled applications.

Application Setups

SHmessages.ini is also used to hold the current setups for the various SpyWorks debugging tools. These entries are set by the individual applications and should not need to be modified by hand.

Message Interpretation

When SpyMsg detects a windows message, it receives an integer message number. Standard messages (those numbered below WM_USER which is defined as &H400) all have the same name regardless of window class. Messages over &H400 may have different names depending on the class of the window.

The **ClassMessages** topic defines the group names which should be given priority for a given class. For example, the entry:

ComboBox=ComboBox

Indicates that classes **ComboBox** and **ThunderComboBox** should first search the **ComboBox** group when choosing the message name for a particular message. Note that SpyMsg automatically strips off 'Thunder' from class names for the purposes of this search.

A list can be specified here as well. If you had a control that had messages from two different groups, you could specify:

classname = group1,group2

to specify the order in which groups should be searched.

After the specified groups are searched, the entire message list will be searched in the order specified by the **MessageGroups** topic.

Style Interpretation

Every window has a style property which defines the styles of the window. The meaning of some of these style bits is common to all windows, however the meanings of some other style bits are defined by the type of window.

The SpyWin **Detail** command attempts to interpret styles correctly according to the class of the window. In many cases, a control that is subclassed from a standard window control can use the standard window styles.

The **ClassList** topic contains a list of aliases for standard classes. These take the form:

new class=standard class

Where the *new class* is the actual class of the control or window, and the *standard class* represents one of the standard windows classes Button, ComboBox, ListBox, or Edit. The complete source for the SpyWin application is included with SpyWorks, so you can easily extend the interpretation of style bits to add custom styles if you wish.

Message Grouping

Most "Spy" or message viewing type programs divide messages into groups for convenience in defining message filters. SpyWorks takes this a step further by allowing you to define your own groupings for windows messages.

The MessageGroups topic contains a list of all of the message groups. Each entry takes the form:

Description=Groupname

Where the *Description* is the description of the group as it will appear in the various SpyWorks group lists, and *Groupname* is the internal name of the group.

Each *Groupname* forms a topic which contains a list of all of the messages in that group. This list contains the name of the message followed by the message value.

You can add additional groups or assign messages to different groups at will.

Technical Support

For information on customer support and last minute changes, refer to the file readme.rtf file. This file is compatible with Wordpad.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 10 lines in it is non-trivial. SpyWorks is emphatically non-trivial....

SpyWorks has undergone extensive testing to make it as bug free as possible. Nevertheless, it is possible that some have crept through. This is especially true considering the fact that both operating systems and development environments are brand new. Please write or send us a fax if you find one, and include all of the steps needed to reproduce the problem. Also, if there are any files needed to reproduce the error, send them to us on a diskette.

If you have any questions you are also welcome to refer to our Frequently Asked Questions section of our web site.

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 SpyWorks, or any suggestions you may have for new products or custom controls.

Please address all correspondence to:

Desaware, Inc.
3510 Charater Park Drive, Suite 48
San Jose, CA 95136
Telephone: 408/404-4760 Fax: 408/404-4780
Web Site: <http://www.desaware.com>
E-mail: support@desaware.com