

SpyWorks™

Version 8.0
for Visual Studio .NET

by
Desaware, Inc.

Rev 8.0.0 (11/06)

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 © 2006 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 .NET Compiled Files: You may include with your program a copy of the files dwsbc80.ocx, dwshk80dwshk80.ocx, Desaware.shcomponent11.dll, Desaware.shcomponent20.dll, and dwshengine80.dll. You may also distribute DLL files created using the ExportWizard.exe utility programs. You may **not** modify the files listed above in any way.

SpyWorks .NET 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 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

Introduction.....	8
New for version 8.0.....	8
SpyWorks and Visual Studio .NET	9
Subclassing and Hooking.....	9
Function Export	9
Known issues with Visual Studio .NET and other general comments	9
File Descriptions	10
Compatibility Issues.....	11
Migrating SpyWorks 7.1 Projects to SpyWorks 8.0.....	11
Migrating Visual Basic 6.0 Projects to .NET	11
Learning .NET	12
Migrating Visual Basic 6.0 projects.....	12
Deciding on which Subclass and WinHook component to use in .NET	12
Migrating SpyWorks Subclass and WinHook ATL based ActiveX controls.....	12
Using SpyWorks (Please Read!).....	12
Customer Support	13
Register! Register! Register!.....	13
SpyWorks Concepts: Subclassing.....	14
Introduction to Subclassing.....	14
Windows Functions	15
How might you use subclassing?.....	16
Cautions on Using Subclassing.....	18
Delayed Events - Posting an Event to Yourself.....	18
Using the Desaware Subclasser	18
Subclassing and spyware	18
Using the Desaware.SpyWorksDotNet Subclasser object.....	19
Using the dwsbc80.ocx control.....	21
Subclassing Multiple Windows with the dwsbc80.ocx control	22
CrossProcess Issues	22
Process Spaces	23
SpyWorks Concepts: Windows Hooks.....	23
Types of hooks.....	25
Should you use hooks or subclassing?.....	27
You are only interested in messages going to one or two windows	27
You are interested in monitoring messages to a large group of windows, such as all of the controls on a form.....	27
You are interested in responding to particular messages regardless of which application is currently active.	27
In general:	27
Using the Desaware Windows Hook	27
Keyboard hooks and spyware	27
Using the Desaware.SpyWorksDotNet KeyHook object for Keyboard Hooks	28
Setting up the KeyHook object for Keyboard Hooks	28
Discarding keystrokes.....	30
Using the dwshk80.ocx control for Keyboard Hooks.....	30

Setting up the dwshk80.ocx control for Keyboard Hooks	31
Key Value Format.....	31
Discarding keystrokes.....	31
Using the Desaware.SpyWorksDotNet WinHook object for Windows Hooks.....	31
Setting up the WinHook object for Windows Hooks	32
Using the dwshk80.ocx control for Windows Hooks	33
Setting up the dwshk80.ocx control for Windows Hooks	34
Use of the nodef Parameter for the dwshk80.ocx control.....	34
WinHook - Use of the nodef event parameter	35
Hook Examples.....	35
For further information on Hooks.....	46
SpyWorks Concepts: dwshengine80.dll function library	47
dwshengine80.dll function reference	47
dwCopyData	47
dwGetAddressForObject.....	47
dwXAllocateDataFrom.....	48
dwXFreeDataFrom	48
dwXCopyAnsiStringFrom	48
dwXCopyUnicodeStringFrom	49
dwXCopyDataTo	49
dwXCopyDataFrom.....	49
dwXGetModuleFileName.....	50
dwXGetEditLine	50
dwXSetForegroundWindow	50
Application Note: Using Cross Process Memory Access with SpyWorks.....	51
Using the EM_GETTEXTRANGE message.....	51
In Process Example.....	53
Cross Process Example.....	55
Conclusion	60
SpyWorks Concepts: Exporting Functions.....	61
What are Exported Functions?.....	61
How Dynamic Export Technology Works	62
The Exports Class	63
The ExportWizard.....	64
Testing Exported Functions	65
Distributing your Exported Function files	65
Warning! Exporting Functions is Dangerous!	65
Migrating to the Desaware.shcomponent.dll	65
Introduction.....	65
Fundamental Differences between the Desaware.shcomponent.dll component and the COM based subclassing and hook components.....	66
The Desaware.shcomponent.dll component	66
Major Changes to the Subclassing Component	67
Major Changes to the Windows hook Component.....	68
Major Changes to the Keyboard hook Component.....	68
Migrating the Subclass control from a .NET project.....	68

Property changes:.....	70
Method changes:.....	72
Event changes:.....	72
Migrating the WinHook control from a .NET project – Windows hook migration	73
Property changes:.....	74
Event changes:.....	76
Migrating the WinHook control from a .NET project – KeyBoard hook migration	78
Property changes:.....	80
Event changes:.....	80
Desaware.shcomponent.dll Reference	81
Introduction.....	81
Desaware.SpyWorks Enumerators	81
Desaware.SpyWorks Minor Classes.....	84
Desaware.SpyWorks Main Classes	103
Controller Class	103
Properties	103
Methods.....	104
KeyHook Class	105
Properties	105
Events.....	106
Subclasser Class.....	108
Properties	109
Events.....	109
WinHook Class	111
Use of the nodef event parameter	111
Properties	111
Events.....	114
dwsbc80.ocx Reference	121
Introduction.....	121
Features:.....	121
Properties	122
Methods.....	127
Events.....	128
dwshk80.ocx Reference	130
Introduction.....	130
Keyboard hook features:.....	130
Keyboard hook Properties.....	130
Keyboard hook Methods.....	134
Keyboard hook Events.....	135
Windows hook features:.....	138
Windows hook Properties.....	138
Windows hook Events	147
.NET samples.....	152
Differences between C# and Visual Basic .NET sample projects	153

Introduction

SpyWorks is probably the most unusual add-on product available for Visual Studio. 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:

1. Like any generic subclassing/hook tool, it can be used by unscrupulous developers to create various types of spyware.
2. We wanted to substantially improve the handling of hooks under adverse situations – such as dealing with system crashes.
3. This release provides initial support for .NET 2.0/3.0.

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 no longer supports the .Net 1.0 framework. Support for .NET 1.0 continues with version 7.1 which is still available and included with Universal COM.
- SpyWorks 8.0 includes a .NET 2.0 subclassing/hook component that is also compatible with .NET 3.0.
- SpyWorks 8.0 does not include primary interop assemblies or examples for using the ActiveX controls with .NET. You can use them in .NET if you wish, however you will need to use the interop assembly generated by Visual Studio.
- SpyWorks 8.0 does not install any components in the GAC. We feel it is now better for .NET assemblies to be distributed in private directories. Note, however, that the dwshengine80.dll component, like the dwspy36.dll component before it, must be installed in the system directory. Simultaneously loading two instances of these DLLs will cause errors to occur.
- 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.

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.

- This release of SpyWorks supports Microsoft Visual Studio .NET versions 1.1 and .NET 2.0/3.0.

Subclassing and Hooking

Support is provided using the native .NET Desaware.shcomponent.dll assembly file, and the dwshk80.ocx and dwsbc80.ocx ActiveX controls.

We are confident that those of you who need the kinds of low level system access provided by these components will be very pleased with their behavior under .NET.

We are also pleased to provide a number of sample .NET programs that demonstrate the use of these components under .NET.

These samples are installed in the “VS NET Samples”, under the SpyWorks main folder.

Function Export

SpyWorks Professional includes a .NET function exporter that allows you to export functions from your .NET assemblies. Other development platforms can call your .NET export functions just as if they would call any standard Windows API functions. Refer to the Exporting Functions section in this manual or the .NET Function Export samples for more details.

Known issues with Visual Studio .NET and other general comments

- Upgrading a Visual Basic 6.0 project into .NET will not upgrade the Subclass or WinHook controls correctly. You would need to change a couple of lines of code in the upgraded file. Refer to the Migrating to .NET from Visual Basic 6.0 help topic for more details.
- The Messages and Keys properties for the Subclass and WinHook controls are also not preserved when upgrading to a Visual Basic .NET project from a Visual Basic 6.0 project. For these properties, you should copy them from the VB 6 project and manually enter them in the .NET project. Also, the value for these properties will appear as “0” in the Property Window when that property is not selected. Once you select the property, you will see a “...” on the entry which you may click to bring up the Property Page for the control.
- There is no automatic way to upgrade from the dwsbc36.ocx or dwshk36.ocx components to dwsbc80.ocx and dwshk80.ocx. The easiest way to switch is to drop a new control on the form and carefully copy the property values from one to the other (using care to copy the keys and messages properties). It is not enough to modify the project files because the binary storage format used for some of the properties has changed.

-
- Many of the dwGetAddress* functions will still work in .NET but we recommend that you use the .NET framework's Marshal namespace and platform-invoke functionality instead. Refer to the function export samples for the Visual Basic 6.0 edition and Visual Studio .NET editions to see how to substitute the functions.
- For those moving to Visual Studio .NET from Visual Basic 6.0, we also recommend Dan Appleman's "Moving to VB.NET: Strategies, Concepts and Code" book for the intermediate to advanced Visual Basic 6 developers <http://www.desaware.com/MovingToVBNETL2.htm>

File Descriptions

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.

Desaware.shcomponent.dll – Main SpyWorks Windows Hook and Subclassing component for Visual Studio .NET. We recommend that you use this component when developing in Visual Studio .NET. This file is installed to your SpyWorks folder's bin subfolder.

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, and dwsbc80.ocx. This file is installed in your System folder.

The following files may NOT be redistributed.

ExportWizard11.exe – Function Export Wizard. This file is installed in your SpyWorks "VS NET Apps\Export" folder.

dwNetExp11.xft – Function Export Wizard dependency file. This file is installed in your SpyWorks "VS NET Apps\Export" folder.

Sw7help.dll – Function Export Wizard dependency file. This file is installed in your System folder.

dwNetExportDiag.exe – Function Export diagnostics tool. This file is installed in your SpyWorks "VS NET Apps\Export" folder.

dwsdes32.dll – SpyWorks Professional license file. This file is installed in your System folder.

SpyWorks includes sample file projects for Visual Studio 1.1 (2003) in Visual Basic .NET and C# formats. The installation program will install the samples to the “VS NET Samples” folder below the main SpyWorks folder.

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 include components based on ActiveX technology and the .NET framework and are compatible with the versions of Visual Basic and Visual Studio .NET that supports these technologies.

The SpyWorks 8.0 components have been tested with Visual Studio .NET versions 1.1 under Windows 2000, Windows XP, and Windows 2003. The sample code and utilities provided are distributed in Visual Basic .NET and C# formats.

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 components 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.

Migrating SpyWorks 7.1 Projects to SpyWorks 8.0

SpyWorks 8.0 includes replacement components as follows:

- dwspy36.dll is replaced by dwshengine80.dll
- desaware.spyworksdotnet11.dll is replaced by desaware.shcomponent11.dll
- There is no automatic way to upgrade from the dwsbc36.ocx or dwshk36.ocx components to dwsbc80.ocx and dwshk80.ocx. The easiest way to switch is to drop a new control on the form and carefully copy the property values from one to the other (using care to copy the keys and messages properties). It is not enough to modify the project files because the binary storage format used for some of the properties has changed.

The desaware.shcomponent11.dll component is functionally identical to desaware.spyworksdotnet11.dll. All you need to do is change the reference in your project from one to the other. You cannot reference both components at the same time.

Migrating Visual Basic 6.0 Projects to .NET

Migrating a Visual Basic 6.0 project to .NET is not recommended. But, nevertheless here are some tips and known issues if you must migrate an existing Visual Basic 6 project that contains SpyWorks to .NET. Please refer to the *Migrating to the Desaware.shcomponent.dll* section for information on migrating .NET projects that uses the SpyWorks ActiveX controls to the Desaware.shcomponent.dll .NET assembly component.

Learning .NET

One of the first steps is learning Visual Studio .NET. Not just the syntax changes or learning the namespaces, but what is really important to learn and leveraging what you already know. For all this, we recommend Dan Appleman's "Moving to VB.NET: Strategies, Concepts, and Code" book, ISBN 1-893115-976 published by Apress. If you are trying to decide whether to stick with the Visual Basic language or go to C#, we recommend Dan Appleman's "Visual Basic.NET or C#? Which to Choose?" ebook which can be purchased directly from Desaware, Inc., or on Amazon.com.

Migrating Visual Basic 6.0 projects

We recommend that you use the .NET upgrade wizard to upgrade your current VB 6 projects to a Visual Basic .NET project. You will still need to make some code modifications after the upgrade wizard finishes porting your project, but at least it does a pretty good job with most of your code migration.

Deciding on which Subclass and WinHook component to use in .NET

We strongly recommend you use the new native .NET Desaware.SpyWorksDotNet namespace (Desaware.shcomponent11.dll DLL) that includes similar objects to replace the Subclass and WinHook ActiveX controls. You can continue to use the ActiveX controls in your .NET project. The initial investment in learning the new objects contained in the Desaware.SpyWorksDotNet namespace will be a little higher, but their similarities to the previous ActiveX controls and .NET objects will make their selection a good investment for future .NET development. If you choose to use the new Desaware.SpyWorksDotNet component, we recommend that you still migrate the Subclass or Windows Hook controls first.

Migrating SpyWorks Subclass and WinHook ATL based ActiveX controls

There are two approaches you can take:

1. First migrate your VB6 project to use the new dwsbc80.ocx and dwshk80.ocx control, then run the upgrade wizard.
2. Install SpyWorks 7.1 and migrate to the older dwsbc36.ocx and dwshk36.ocx control. Refer to the SpyWorks 7.1 documentation for further details on this migration.

The Messages and Keys properties will not be migrated by the upgrade wizard.

Using SpyWorks (Please Read!)

SpyWorks is designed for the intermediate to advanced Visual Basic or C# developer who has a knowledge of how to use the Windows Application Programmer's Interface (API). Also, 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++, and

then write it in Visual Basic or C#. Any code that you would normally write in a Windows procedure in response to a Windows message, you can place in the SpyWorks Subclass event. Any time you need to export a function from a DLL, look at the Exports class functionality provided by SpyWorks. Where you would use Windows hooks, use the SpyWorks Windows Hook control or WinHook object.

If you have never programmed in Windows, you must learn about it in order to use this package effectively. Visual Studio includes a reference for all available Windows API functions and Windows messages.

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 Studio. 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 and Windows messages. 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 Studio.

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 phone 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 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 provide you with technical support, send you information about upgrades, or send you upgrades if you have a firewall preventing the auto update from automatically retrieving update files. 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 Charter Park Drive, 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 list server by sending a message to listserv@desaware.com and including the word "Subscribe" in the subject line. We do not share or sell your email address and 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. To handle the entire spectrum of subclassing requirements, SpyWorks includes two different approaches to subclassing. The dwsbc80.ocx subclass control is the ATL-based ActiveX control that was used in previous versions of Visual Basic and supported in .NET. The Desaware.shcomponent.dll component is a native .NET assembly that is in many ways more efficient than the ActiveX control.

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:

ReturnValue 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 bits under Win32. The ReturnValue is a 32 bit value. 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.

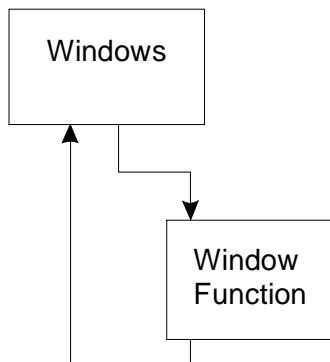


Figure 1
Windows Functions

The SpyWorks subclassing components supports 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.

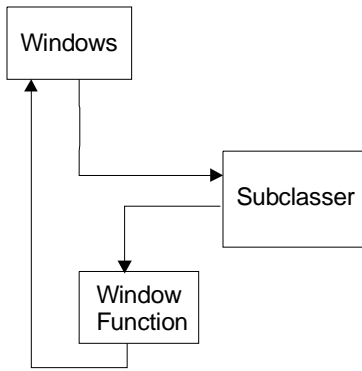


Figure 2
Subclassing before default processing

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.

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.

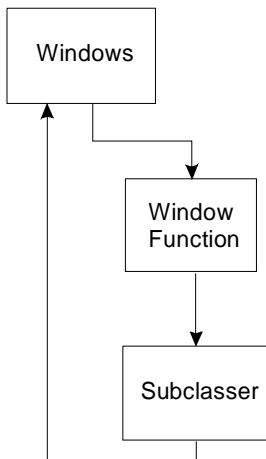


Figure 3
Subclassing after default processing

You can also indicate that specific messages should be posted to the SpyWorks subclassing components (posted or asynchronous 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.

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 .NET 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.

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 can see, choosing the type of subclassing is a critical decision. You should always use Posted (or Asynchronous) 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 dwsbc.ocx control or the SubclassingType property of the Desaware.SpyWorksDotNet Subclasser 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 or asynchronous), 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 Diagnostics.Debug 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 (or asynchronous) 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.

The recommended approach to doing this is to use an asynchronous delegate.

If you are using the ActiveX control, 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.

Using the Desaware Subclasser

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 the Desaware.SpyWorksDotNet Subclasser object

Subclassing using the native .NET Desaware.SpyWorksDotNet Subclasser object is a very simple process.

- Add the Desaware.shcomponent.dll .NET component reference to your .NET project. The Desaware.shcomponent.dll file is installed in your SpyWorks bin folder.
- Define the Subclasser object and create a new instance of the object.

[VB]

```
Imports Desaware.SpyWorks
```

```
Friend SubClass1 As Subclasser  
SubClass1 = New Subclasser()
```

[C#]

```
using Desaware.SpyWorks;
```

```
internal Subclasser SubClass1;  
SubClass1 = new Subclasser();
```

- Specify the messages to detect. The Subclasser object only detects messages that you specify. This helps keep the overhead in subclassing to an absolute minimum. Use the *Messages* property to specify the messages to detect. If you do not specify any messages, the object will detect all messages going to the subclassed window.

[VB]

```
Imports Desaware.SpyWorks
```

```
SubClass1.Messages = New WindowsMessageList()  
SubClass1.Messages.AddMessage(StandardMessages.WM_ACTIVATE)
```

[C#]

```
using Desaware.SpyWorks;
```

```
SubClass1.Messages = new WindowsMessageList();  
SubClass1.Messages.AddMessage(StandardMessages.WM_ACTIVATE);
```

- 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 asynchronously (posted).

[VB]

```
SubClass1.SubclassingType = SubclassingTypes.PreDefault
```

[C#]

```
SubClass1.SubclassingType = SubclassingTypes.PreDefault;
```

- Create an event handler for the Subclasser object's *OnWndMessage* event. (You can also declare the subclass object *WithEvents* in VB .NET).

[VB]

```
AddHandler SubClass1.OnWndMessage, AddressOf SubClass1_OnWndMessage
```

```
Private Sub SubClass1_OnWndMessage(ByVal sender As Object, ByVal e As  
Desaware.SpyWorks.WndMessageEventArgs)
```

```
End Sub
```

[C#]

```
SubClass1.OnWndMessage += new  
WndMessageEventHandler(SubClass1_OnWndMessage);
```

```
private void SubClass1_OnWndMessage(object sender,  
Desaware.SpyWorks.WndMessageEventArgs e)  
{  
}
```

- For cleanup purposes, the Subclasser object's *OnWndMessage* event handler should also be removed when you are done using the object.

[VB]

```
RemoveHandler SubClass1.OnWndMessage, AddressOf SubClass1_OnWndMessage
```

[C#]

```
SubClass1.OnWndMessage -= new  
WndMessageEventHandler(SubClass1_OnWndMessage);
```

- Set the *HwndParam* property to the window, control or form to subclass. Subclassing starts immediately after a window is specified so this should be the last step performed. To end subclassing, clear the *HwndParam* property.

[VB]

```
SubClass1.HwndParam = hwnd
```

[C#]

```
SubClass1.HwndParam = hwnd;
```

Once you have performed these steps, the Subclasser object will receive messages based on the property settings and will trigger the *OnWndMessage* event.

The windows message information is exposed by the *OnWndMessage* event's *WndMessageEventArgs* parameter. The window handle is in the *e.hwnd* parameter. The message number can be found in the *e.msg* parameter. *e.wp* and *e.lp* are the standard windows *wParam* and *lParam* parameters, and their values depend on the individual message. If you are doing asynchronous (or 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 *e.nodef* parameter to non-zero, you can block the default message processing from taking place and specify your own return value by setting the *e.retval* parameter.

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

The *e.retval* and *e.nodef* parameters have no effect when the subclassing type is asynchronous.

Using the *dwsbc80.ocx* control

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

- Add the *dwsbc80.ocx* control to your Windows form. If the *dwsbc80.ocx* control is not already in your Toolbox you can right-click on the Toolbox and select the *Customize Toolbox...* menu command. In the *COM Components* tab of the *Customize Toolbox* form, select the *Desaware dwsbc80 v8 Subclassing Control* checkbox then select the OK button to add the *dwsbc80* control to your Toolbox.
- Select the messages to detect. The *dwsbc80.ocx* control 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.
- 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
- 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. Subclassing starts immediately after a window is specified so this should be the last step performed.

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 appears as follows:

```
[VB]
```

```
Private Sub SubClass1_WndMessage(ByVal sender As Object, ByVal e As
AxDWSHK80Lib.AxSubclass._DDwsbcEvents_WndMessageEvent) Handles
SubClass1.WndMessage
[C#]
private void SubClass1_WndMessage(object sender,
AxDWSHK80Lib.AxSubclass._DDwsbcEvents_WndMessageEvent e)
```

The window handle is in the *e.hwnd* parameter. The message number can be found in the *e.msg* parameter. *e.wp* and *e.lp* are the standard windows *wParam* and *lParam* parameters, and their values depend on the individual message. If you are doing a posted (or asynchronous) 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 *e.nodef* parameter to non-zero, you can block the default message processing from taking place and specify your own return value by setting the *e.retval* parameter. If you are using post-default processing, the *e.retval* parameter will already be set to the return value provided by the default window message processing. The *e.retval* and *e.nodef* parameters have no effect when the subclassing type is *Posted*.

Subclassing Multiple Windows with the *dwsbc80.ocx* control

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.

CrossProcess Issues

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 ask for.)

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? 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 SpyWorks Subclassing components, where all of the messages in the system can become blocked while waiting for a single application. For this reason, the SpyWorks Subclassing and Windows Hook components 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. Note that the Desaware.shcomponent.dll native component is more resistant to this kind of deadlock because it can take advantage of the free threaded nature of .NET, especially when using asynchronous message detection.

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-process memory function in the dwshengine80.dll library (more on this later). However, the SpyWorks Subclassing components also include the GetAnsiString and GetUnicodeString methods to allow you to easily retrieve text information from other process spaces.

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 main 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.

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.

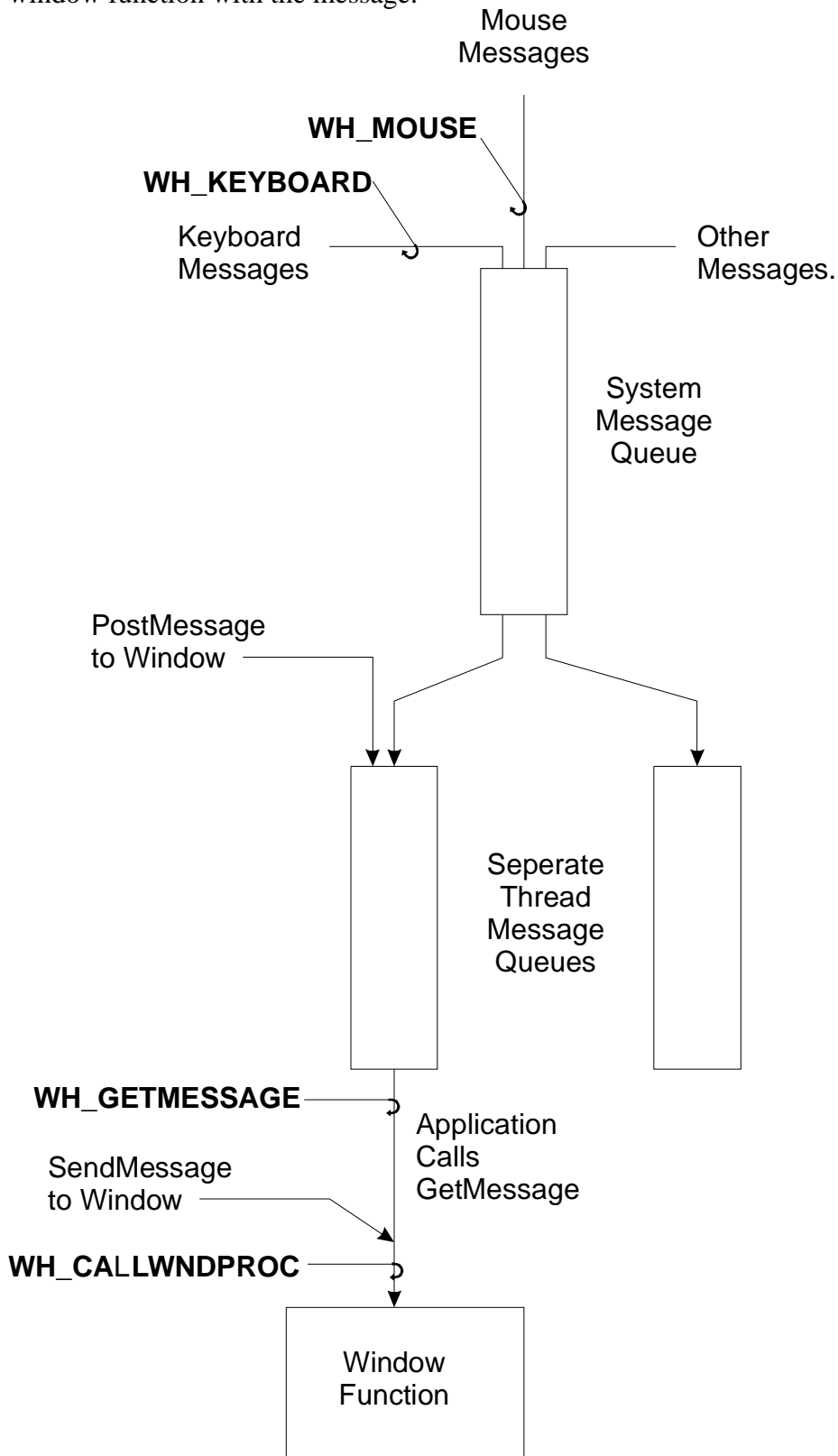


Figure 4 - Message flow and Hooks

The SendMessage and GetMessage message paths are both shown at the bottom of figure 4. The figure illustrates the first 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 dwshk80.ocx control is the ATL-based ActiveX control that was used in previous versions of Visual Basic and supported in .NET. The Desaware.shcomponent.dll component is a native .NET assembly that is in many ways more efficient than the ActiveX control.

Types of hooks

The SpyWorks Windows hooks components support most current types of Windows hooks. Of these, the most likely ones that you will use are the WH_GETMESSAGE (GetMessage), WH_MOUSE (Mouse), WH_KEYBOARD (Keyboard) and WH_CALLWNDPROC (CallWndProc) hooks. Refer to the online component reference and the HookType properties of the dwshk80.ocx control or WinHook object for details on how to use these hook types. Note that the dwshk80.ocx control and WinHook object raise different events for different types of hooks. This is also covered in the online documentation for the HookType property.

CallWndProc	Implements a WH_CALLWNDPROC hook. This hook is
-------------	--

	triggered any time a message is sent to a window function. This hook type detects every message received by a window. Even with the advanced filtering used by SpyWorks, use of this hook can impact system performance and should be avoided if possible.
CallWndProcRet	Implements a WH_CALLWNDPROCRET hook. This hook is triggered any time a windows function returns from a message. This hook type detects every message received by a window. Even with the advanced filtering used by SpyWorks, use of this hook can impact system performance and should be avoided if possible.
CBT	Implements a WH_CBT hook. This hook is used to implement computer based training applications, providing information on a variety of windows events.
ForegroundIdle	Implements a WH_FOREGROUNDIDLE hook. This hook is used to detect when the foreground thread is about to become idle.
GetMessage	Implements a WH_GETMESSAGE hook. This hook is triggered any time an API function called GetMessage (or PeekMessage) 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.
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.
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.
Keyboard	Implements a WH_KEYBOARD hook. This hook is triggered by keyboard events.
KeyboardLL	Implements a WH_KEYBOARD_LL hook. This hook is triggered by keyboard events.
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.
Mouse	Implements a WH_MOUSE hook. This hook is triggered by mouse events.
MouseLL	Implements a WH_MOUSE_LL hook. This hook is triggered by mouse events.
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.
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.

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 where messages are being blocked by other parts of the system before they get to the window.

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.

Using the Desaware Windows Hook

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 the Desaware.SpyWorksDotNet KeyHook object for Keyboard Hooks

The KeyHook object derives from the WinHook object and is used for keyboard hooks. The KeyHook object 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 HookType and Monitor properties. One type intercepts only keystrokes sent to the process that contains this object. 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. Refer to the HookType property for more details. Keyboard hooking can be enabled or disabled by setting the Enabled property. Keystrokes may be processed immediately by the application, or posted for later use. The KeyFilterList property can be used to set up a filter for keystroke processing. If no keys are specified, then all keys are detected. Otherwise, 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.

The KeyHook object uses a Windows keyboard hook to detect keyboard events. As such, it detects the keys before they are seen by Visual Studio or any other application. This means that you can detect unusual key combinations such as enter, tab, alt-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.

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

Setting up the KeyHook object for Keyboard Hooks

Receiving keyboard events using the KeyHook object is a very simple process.

1. Add the Desaware.shcomponent.dll reference to your project.

Select the Desaware.shcomponent.dll component from the SpyWorks bin folder.

2. Declare a KeyHook object in the appropriate form, class or module.

Note that you can declare the KeyHook object "WithEvents" in Visual Basic.

[VB]

```
Imports Desaware.SpyWorks
Friend KeyHook1 As KeyHook
```

[C#]

```
using Desaware.SpyWorks;
internal KeyHook KeyHook1;
```

3. Create a new instance of the KeyHook object.

[VB]

```
KeyHook1 = New KeyHook()
```

```
[C#]  
KeyHook1 = new KeyHook();
```

4. Specify the scope of the hook.

The Monitor property specifies the scope of the keyboard hook.

```
[VB]  
KeyHook1.Monitor = HookMonitor.EntireSystem  
[C#]  
KeyHook1.Monitor = HookMonitor.EntireSystem;
```

5. Select the type of hook

The HookType property selects the type of keyboard hook you want to use.

```
[VB]  
KeyHook1.HookType = HookTypes.Keyboard  
[C#]  
KeyHook1.HookType = HookTypes.Keyboard;
```

6. Select the keys to intercept

The default is for the KeyHook object to detect all keystrokes. However, if you are searching only for a specific set of key combinations, you can use the KeyFilterList property to select the keystrokes to intercept. Using a keys filter in this manner will improve performance. The following sample detects the “Ctrl+a” key.

```
[VB]  
KeyHook1.KeyFilterList = New KeyList()  
KeyHook1.KeyFilterList.AddKey(LetterKeys.LTR_A, KeyFlags.Ctrl)  
[C#]  
KeyHook1.KeyFilterList = new KeyList();  
KeyHook1.KeyFilterList.AddKey(LetterKeys.LTR_A, KeyFlags.Ctrl);
```

7. Add event handler

You can add an event handler to the KeyHook’s OnKeyDown or OnKeyUp events depending on when you would like to detect the key.

```
[VB]  
AddHandler KeyHook1.OnKeyDown, AddressOf KeyHook1_OnKeyDown  
[C#]  
KeyHook1.OnKeyDown += new  
Desaware.SpyWorks.KeyboardHookEventHandler(KeyHook1_OnKeyDown);
```

8. Add event code

Add your code to the OnKeyDown or OnKeyUp events.

```
[VB]  
Private Sub KeyHook1_OnKeyDown(ByVal sender As Object, ByVal e As  
Desaware.SpyWorks.KeyboardHookEventArgs)  
    ' Add code here  
End Sub  
[C#]  
private void KeyHook1_OnKeyDown(object sender,  
Desaware.SpyWorks.KeyboardHookEventArgs e)  
{  
    // Add code here  
}
```

9. Start receiving keys

Set the KeyHook object's Enabled property to true to start receiving keys.

```
[VB]
KeyHook1.Enabled = True
[C#]
KeyHook1.Enabled = true;
```

10. Do clean up when done using the KeyHook

Set the KeyHook object's Enabled property to false when done. You should also remove the handler before destroying the KeyHook object.

```
[VB]
KeyHook1.Enabled = False

RemoveHandler KeyHook1.OnKeyDown, AddressOf KeyHook1_OnKeyDown

[C#]
KeyHook1.Enabled = false;

KeyHook1.OnKeyDown -= new
Desaware.SpyWorks.KeyDownHookEventHandler(KeyHook1_OnKeyDown);
```

The KeyboardHookEventArgs object from the OnKeyDown and OnKeyUp events is described in details in the reference section. It includes all the information you need to process the keystroke that was detected.

Discarding keystrokes

The KeyboardHookEventArgs object has two members that are significant when it comes to discarding keystrokes. Setting the *discard* member to true prevents subsequent hooks from seeing the keystroke. But you should also set the *keycode* member to zero to make sure that the original *keycode* is not forwarded to the intended receiver.

Naturally, the AsyncNotification property must be false to detect the keystrokes when hooked. You cannot remove keystrokes if the AsyncNotification property is set.

Using the dwshk80.ocx control for Keyboard Hooks

The dwshk80.ocx control contains two separate subsystems, one for keyboard hooks, the other subsystem for all other types of hooks. You should only enable one of these subsystem per control.

The dwshk80.ocx control 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.

The dwshk80.ocx control uses a Windows keyboard hook to detect keyboard events. As such, it detects the keys before they are seen by Visual Studio 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.

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

Setting up the dwshk80.ocx control 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 the dwshk80.ocx control 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 the dwshk80.ocx control is represented by the *keycode* field. The *shiftstate* field determines 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.

Discarding keystrokes

The keyboard events have two parameters that are significant when it comes to discarding keystrokes. Setting the *discard* field prevents subsequent hooks from seeing the keystroke. But you should also set the *keycode* field 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 the Desaware.SpyWorksDotNet WinHook object for Windows Hooks

The WinHook object provides limited ability to modify or to discard messages. The limitations depend on the types of hook, not the object itself. Unlike subclassing with the Subclasser object, you cannot return a result to Windows.

Because Windows hooks do not require a window handle, it is possible for the WinHook object to detect the WM_NCCREATE and WM_CREATE messages that occur when a window is created. This makes it possible to change the style of a newly loaded Windows form or control during the creation process.

Setting up the WinHook object for Windows Hooks

Receiving windows events using the WinHook object is a very simple process.

1. Add the Desaware.shcomponent.dll reference to your project. (This may be Desaware.shcomponent11.dll or Desaware.shcomponent20.dll depending on framework version)

Select the Desaware.shcomponent.dll component from the SpyWorks bin folder.

2. Declare a WinHook object in the appropriate form, class or module.

Note that you can declare the WinHook object "WithEvents" in Visual Basic.

[VB]

```
Imports Desaware.SpyWorks
Friend WinHook1 As WinHook
```

[C#]

```
using Desaware.SpyWorks;
internal WinHook WinHook1;
```

3. Create a new instance of the WinHook object.

[VB]

```
WinHook1 = New WinHook()
```

[C#]

```
WinHook1 = new WinHook();
```

4. Specify the scope of the hook.

The Monitor property specifies the scope of the windows hook.

[VB]

```
WinHook1.Monitor = HookMonitor.EntireSystem
```

[C#]

```
WinHook1.Monitor = HookMonitor.EntireSystem;
```

5. Select the type of hook

The HookType property selects the type of windows hook you want to use.

[VB]

```
WinHook1.HookType = HookTypes.Mouse
```

[C#]

```
WinHook1.HookType = HookTypes.Mouse;
```

6. Select the messages to detect

The default is for the WinHook object to detect all windows messages. However, if you are searching only for a specific set of windows messages, you can use the

WindowsMessageList property to select the windows messages to detect. Using a

message filter in this manner will improve performance. The following sample detects the mouse movement and left mouse button click over the non-client area of a window.

[VB]

```
WinHook1.Messages = New WindowsMessageList()
```

```
WinHook1.Messages.AddMessage(NonClientMessages.WM_NCMOUSEMOVE)
```

```
WinHook1.Messages.AddMessage(NonClientMessages.WM_NCLBUTTONDOWN)
```

[C#]

```
WinHook1.Messages = new WindowsMessageList();
```

```
WinHook1.Messages.AddMessage(NonClientMessages.WM_NCMOUSEMOVE);
```

```
WinHook1.Messages.AddMessage(NonClientMessages.WM_NCLBUTTONDOWN);
```

7. Add event handler

You can add an event handler to the WinHook object's OnCBTHook, OnForegroundIdleHook, OnJournalPlaybackHook, OnJournalRecordHook, OnMessageHook, OnMouseHook, or OnShellHook events depending on the HookType you select.

```
[VB]
AddHandler WinHook1.OnMouseHook, AddressOf WinHook1_OnMouseHook
[C#]
WinHook1.OnMouseHook += new
Desaware.SpyWorks.MouseHookEventHandler(WinHook1_OnMouseHook);
```

8. Add event code

Add your code to the WinHook object's event. Each event's *e* parameter is of a different object containing information for that particular hook type. Refer to the reference section for more details on each event.

```
[VB]
Private Sub WinHook1_OnMouseHook(ByVal sender As Object, ByVal e As
Desaware.SpyWorks.MouseHookEventArgs)
    ' Add code here
End Sub
[C#]
private void WinHook1_OnMouseHook(object sender,
Desaware.SpyWorks.MouseHookEventArgs e)
{
    // Add code here
}
```

9. Start detecting messages

Set the WinHook object's Enabled property to true to start receiving messages.

```
[VB]
WinHook1.Enabled = True
[C#]
WinHook1.Enabled = true;
```

10. Do clean up when done using the WinHook

Set the WinHook object's Enabled property to false when done. You should also remove the handler before destroying the WinHook object.

```
[VB]
WinHook1.Enabled = False

RemoveHandler WinHook1.OnMouseHook, AddressOf WinHook1_OnMouseHook

[C#]
WinHook1.Enabled = false;

WinHook1.OnMouseHook -= new
Desaware.SpyWorks.MouseHookEventHandler(WinHook1_OnMouseHook);
```

Using the dwshk80.ocx control 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 Windows form or control during the creation process.

Setting up the dwshk80.ocx control for Windows Hooks

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

1. Select the type of hook

Use the HookType 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.

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 it's 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 online reference for the HookType property to see which events are associated with each hook type.

Use of the nodef Parameter for the dwshk80.ocx control

The dwshk80.ocx message events include a nodef field. This field 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.

The dwshk80.ocx control uses a different technology to intercept windows messages. Setting the nodef field 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 dwshk80.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. The dwshk80.ocx control 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 field to True during the message event processing.

WinHook - Use of the nodef event parameter

Most WinHook event argument classes include a nodef field that can be set during event processing. This event provides direction to the dwshengine80.dll engine to not call the CallNextHookEx function. If other applications have placed the same hook, this will in many cases prevent the other hook from being called. If the hook accepts a True return value to indicate the message was handled, setting nodef to non-zero will return True. For CBTHooks, setting nodef to non-zero will cause the value specified by the BlockCBTOperation parameter to be returned.

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. The first implementation is based on using the Desaware.shcomponent.dll WinHook component. The second implementation is based on using the dwshk80.ocx WinHook Control.

SpyWorksDotNet WinHook component implementation

1. Add the Desaware.shcomponent.dll reference to your project. Declare a WinHook object in the appropriate form, class or module. Create a new instance of the WinHook object.

```
[VB]
Imports Desaware.SpyWorks
Friend WinHook1 As WinHook
WinHook1 = New WinHook()
```

```
[C#]
using Desaware.SpyWorks;
internal WinHook WinHook1;
WinHook1 = new WinHook();
```

2. Create a new instance of the WinHook object's WindowsMessageList (Messages property).

```
[VB]
WinHook1.Messages = New WindowsMessageList()
```

```
[C#]
WinHook1.Messages = new WindowsMessageList();
```

3. Using the AddMessage method of the WindowsMessageList object, add the MouseMessages.WM_RBUTTONDOWN message to the messages list.

```
[VB]
WinHook1.Messages.AddMessage(MouseMessages.WM_RBUTTONDOWN)
```

```
[C#]
WinHook1.Messages.AddMessage(MouseMessages.WM_RBUTTONDOWN);
```

4. Set the WinHook object's Monitor property to "HookMonitor.ThisProcess".

[VB]

```
WinHook1.Monitor = HookMonitor.ThisProcess
```

[C#]

```
WinHook1.Monitor = HookMonitor.ThisProcess;
```

5. Set the WinHook object's HookType property to "HookTypes.Mouse".

[VB]

```
WinHook1.HookType = HookTypes.Mouse
```

[C#]

```
WinHook1.HookType = HookTypes.Mouse;
```

6. Create an OnMouseHook function to handle the WinHook object's OnMouseHook event, and connect the event handler to the WinHook object's OnMouseHook event.

[VB]

```
Private Sub WinHook1_OnMouseHook(ByVal sender As Object, ByVal e As  
MouseHookEventArgs)  
    Debug.WriteLine "User right clicked on " + e.hwnd.ToString  
End Sub
```

```
AddHandler WinHook1.OnMouseHook, AddressOf WinHook1_OnMouseHook
```

[C#]

```
private void WinHook1_OnMouseHook(object sender, MouseEventArgs e)  
{  
    Debug.WriteLine "User right clicked on " + e.hwnd.ToString()  
}
```

```
WinHook1.OnMouseHook += new  
MouseHookEventHandler(WinHook1_OnMouseHook);
```

7. To start detecting for the message, set the WinHook object's Enabled property to True.

[VB]

```
WinHook1.Enabled = True
```

[C#]

```
WinHook1.Enabled = true;
```

8. When finished using the WinHook object, set the Enabled property to False to stop the message processing. Before deleting the object, disconnect the OnMouseHook event.

[VB]

```
WinHook1.Enabled = False  
RemoveHandler WinHook1.OnMouseHook, AddressOf WinHook1_OnMouseHook
```

[C#]

```
WinHook1.Enabled = false;  
WinHook1.OnMouseHook -= new  
MouseHookEventHandler(WinHook1_OnMouseHook);
```

dwshk80.ocx WinHook Control implementation

1. Add the Desaware SpyWorks Windows Hook Control component to your project. Add a WinHook control to Form1. To add a reference to the Hook control, select the Desaware dwshk80 v8 Hook Control from the COM tab of the Add Reference form. To add the Hook control to your form, you may need to add it to your toolbox first by using the Customize Toolbox command and adding the Desaware dwshk80 v8 hook Hook Control from the COM Components tab of the Customize Toolbox form.
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.
3. Set the WinHook control's Monitor property to "4 – This Task".
4. Set the WinHook control's HookType property to "1 - WH_MOUSE".
5. In Form_Load event of Form1, set the WinHook control's HookEnabled property to True.
6. Attach the following code to the WinHook control's MouseProc.

[VB]

```
Debug.WriteLine "User right clicked on " + e.wnd.ToString()
```

[C#]

```
Debug.WriteLine "User right clicked on " + e.wnd.ToString();
```

- This example demonstrates how to monitor the entire system to determine application switching. The first implementation is based on using the Desaware.shcomponent.dll WinHook component. The second implementation is based on using the dwshk80.ocx WinHook Control.

SpyWorksDotNet WinHook component implementation

1. Add the Desaware.shcomponent.dll reference to your project. Declare a WinHook object in the appropriate form, class or module. Create a new instance of the WinHook object.

[VB]

```
Imports Desaware.SpyWorks  
Friend WinHook1 As WinHook  
WinHook1 = New WinHook()
```

[C#]

```
using Desaware.SpyWorks;  
internal WinHook WinHook1;  
WinHook1 = new WinHook();
```

2. Create a new instance of the WinHook object's WindowsMessageList (Messages property).

[VB]

```
WinHook1.Messages = New WindowsMessageList()
```

[C#]

```
WinHook1.Messages = new WindowsMessageList();
```

- Using the `AddMessage` method of the `WindowsMessageList` object, add the `StandardMessages.WM_ACTIVATEAPP` message to the messages list.

[VB]

```
WinHook1.Messages.AddMessage(StandardMessages.WM_ACTIVATEAPP)
```

[C#]

```
WinHook1.Messages.AddMessage(StandardMessages.WM_ACTIVATEAPP);
```

- Set the `WinHook` object's `Monitor` property to `"HookMonitor.EntireSystem"`.

[VB]

```
WinHook1.Monitor = HookMonitor.EntireSystem
```

[C#]

```
WinHook1.Monitor = HookMonitor.EntireSystem;
```

- Set the `WinHook` object's `HookType` property to `"HookTypes.CallWndProc"`.

[VB]

```
WinHook1.HookType = HookTypes.CallWndProc
```

[C#]

```
WinHook1.HookType = HookTypes.CallWndProc;
```

- Create an `OnMessageHook` function to handle the `WinHook` object's `OnMessageHook` event, and connect the event handler to the `WinHook` object's `OnMessageHook` event. Add code to the `OnMessageHook` event to indicate that another application has been activated.

[VB]

```
Private Declare Auto Function GetWindowThreadProcessId Lib "User32"  
(ByVal hwnd As Integer, ByRef lpdwProcessId As Integer) As Integer
```

```
Private Sub WinHook1_OnMessageHook(ByVal sender As Object, ByVal e As  
MessageHookEventArgs)
```

```
    Static currentprocessid As Integer
```

```
    Dim threadid, processid As Integer
```

```
    If e.wp Then
```

```
        threadid = GetWindowThreadProcessId(e.hwnd, processid)
```

```
        If processid <> currentprocessid Then
```

```
            currentprocessid = processid
```

```
            Debug.WriteLine "Active process changed to " &
```

```
Hex$(currentprocessid)
```

```
        End If
```

```
        Debug.WriteLine Hex$(e.hwnd) & " window activated, thread " &
```

```
Hex$(e.lp) & " is de-activated"
```

```
    Else
```

```
        Debug.WriteLine Hex$(e.hwnd) & " window de-activated, thread "
```

```
& Hex$(e.lp) & " activated"
```

```
    End If
```

```
End Sub
```

```
AddHandler WinHook1.OnMessageHook, AddressOf WinHook1_OnMessageHook
```

[C#]

```

[DllImport("user32.dll")] internal static extern int
GetWindowThreadProcessId (int hwnd, ref int lpdwProcessId);
int currentprocessid;

private void WinHook1_OnMessageHook (object sender,
MessageHookEventArgs e)
{
    int threadid, processid = 0;

    if (e.wp != 0)
    {
        threadid = GetWindowThreadProcessId(e.hwnd.ToInt32(), ref
processid);

        if (processid != currentprocessid)
        {
            currentprocessid = processid;
            Debug.WriteLine("Active process changed to " +
currentprocessid.ToString("x"));
        }

        Debug.WriteLine(e.hwnd.ToInt32().ToString("x") + " window
activated, thread " + e.lp.ToString("x") + " is de-activated");
    }

    else
        Debug.WriteLine(e.hwnd.ToInt32().ToString("x") + " window de-
activated, thread " + e.lp.ToString("x") + " activated");
}

WinHook1.OnMessageHook += new
MessageHookEventHandler(WinHook1_OnMessageHook);

```

7. To start detecting for the message, set the WinHook object's Enabled property to True.

```

[VB]
WinHook1.Enabled = True

```

```

[C#]
WinHook1.Enabled = true;

```

8. When finished using the WinHook object, set the Enabled property to False to stop the message processing. Before deleting the object, disconnect the OnMessageHook event.

```

[VB]
WinHook1.Enabled = False
RemoveHandler WinHook1.OnMessageHook, AddressOf WinHook1_OnMessageHook

```

```

[C#]
WinHook1.Enabled = false;
WinHook1.OnMessageHook -= new
MessageHookEventHandler(WinHook1_OnMessageHook);

```

dwshk80.ocx WinHook Control implementation

1. Add the Desaware SpyWorks Windows 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_ACTIVATEAPP message from the Standard message group to the Selected Messages list. We recommend Daniel Appleman's Visual Basic Programmer's Guide to the Win32 API book as a reference guide to the Windows messages.
3. Set the WinHook control's HookType property to "4 – WH_CALLWNDPROC"
4. Set the WinHook control's Monitor property to "6 – Entire System".
5. In Form_Load event of Form1, set the WinHook control's HookEnabled property to True.
6. Declare the GetWindowThreadProcessId API function and 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).

[VB]

```
Private Declare Auto Function GetWindowThreadProcessId Lib "User32"
    (ByVal hwnd As Integer, ByRef lpdwProcessId As Integer) 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. In our case, we just want to detect when an application has
' been switched.
Private Sub WinHook1_WndMessage(ByVal sender As Object, ByVal e As
    AxDWSHK80Lib._DDwshkEvents_WndMessageEvent) Handles WinHook1.WndMessage
    Static currentprocessid As Integer
    Dim threadid, processid As Integer

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

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

[C#]

```
[DllImport("user32.dll")] internal static extern int
GetWindowThreadProcessId (int hwnd, ref int lpdwProcessId);

int currentprocessid;

// 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.
private void WinHook1_WndMessage (object sender,
AxDWSHK80Lib._DDwshkEvents_WndMessageEvent e)
{
    int threadid, processid = 0;

    if (e.wp != 0)
    {
        threadid = GetWindowThreadProcessId(e.wnd, ref processid);

        if (processid != currentprocessid)
        {
            currentprocessid = processid;
            Debug.WriteLine("Active process changed to " +
currentprocessid.ToString("x"));
        }

        Debug.WriteLine(e.hwnd.ToString("x") + " window activated,
thread " + e.lp.ToString("x") + " is de-activated");
    }

    else
        Debug.WriteLine(e.hwnd. ToString("x") + " window de-
activated, thread " + e.lp.ToString("x") + " activated");
}

```

This example demonstrates how to detect the Enter key and substitute the Tab key in its place. This is useful if you have an Accept 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) under certain circumstances.

SpyWorksDotNet WinHook component implementation

1. Add the Desaware.shcomponent.dll reference to your project. Declare a KeyHook object in the appropriate form, class or module. Create a new instance of the KeyHook object.

```

[VB]
Imports Desaware.SpyWorks
Friend KeyHook1 As KeyHook
KeyHook1 = New KeyHook()

```

```

[C#]
using Desaware.SpyWorks;
internal KeyHook KeyHook1;
KeyHook1 = new KeyHook();

```

2. Create a new instance of the KeyHook object's KeyList (KeyFilterList property) object.

```

[VB]
KeyHook1.Messages = New KeyFilterList()

```

```

[C#]
KeyHook1.Messages = new KeyFilterList();

```

- Using the `AddKey` method of the `KeyList` object, add the `VirtualKeys.VK_Enter` key to the keys list.

[VB]

```
KeyHook1.KeyFilterList.AddKey(VirtualKeys.VK_Enter, KeyFlags.None)
```

[C#]

```
KeyHook1.KeyFilterList.AddKey(VirtualKeys.VK_Enter, KeyFlags.None);
```

- Set the `KeyHook` object's `Monitor` property to `"HookMonitor.ThisProcess"`.

[VB]

```
KeyHook1.Monitor = HookMonitor.ThisProcess
```

[C#]

```
KeyHook1.Monitor = HookMonitor.ThisProcess;
```

- Set the `KeyHook` object's `HookType` property to `"HookTypes.Keyboard"`.

[VB]

```
KeyHook1.HookType = HookTypes.Keyboard
```

[C#]

```
KeyHook1.HookType = HookTypes.Keyboard;
```

- Create an `OnKeyDown` function to handle the `KeyHook` object's `OnKeyDown` event, and connect the event handler to the `KeyHook` object's `OnKeyDown` event. Add code to the `OnKeyDown` event to discard the Enter key and send the Tab key.

[VB]

```
AddHandler WinHook1.OnKeyDown, AddressOf WinHook1_OnKeyDown
```

```
Private Sub WinHook1_OnKeyDown (ByVal sender As Object, ByVal e As  
KeyboardHookEventArgs)
```

```
    e.discard = True
```

```
    SendKeys.Send("{TAB}")
```

```
End Sub
```

[C#]

```
KeyHook1.OnKeyDown += new KeyDownHookEventHandler(KeyHook1_OnKeyDown);
```

```
private void KeyHook1_OnKeyDown(object sender, KeyboardHookEventArgs e)
```

```
{
```

```
    e.discard = true;
```

```
    SendKeys.Send("{TAB}");
```

```
}
```

- To start detecting for the message, set the `KeyHook` object's `Enabled` property to `True`.

[VB]

```
KeyHook1.Enabled = True
```

[C#]

```
KeyHook1.Enabled = true;
```

- When finished using the KeyHook object, set the Enabled property to False to stop the message processing. Before deleting the object, disconnect the OnKeyDown event.

[VB]

```
KeyHook1.Enabled = False
RemoveHandler KeyHook1.OnKeyDown, AddressOf KeyHook1_OnKeyDown
```

[C#]

```
KeyHook1.Enabled = false;
KeyHook1.OnKeyDown -= new KeyDownHookEventHandler(KeyHook1_OnKeyDown);
```

dwshk80.ocx WinHook Control implementation

- Add the Desaware SpyWorks Windows Hook Control component to your project. Add a WinHook control to Form1. To add a reference to the Hook control, select the Desaware dwshk80 v8 Hook Control from the COM tab of the Add Reference form. To add the Hook control to your form, you may need to add it to your toolbox first by using the Customize Toolbox command and adding the Desaware dwshk80 v8 Hook Control from the COM Components tab of the Customize Toolbox form.
- Select the WinHook control's *Keys* property to display the Select Keys form. Select the Enter key from the Available Keys list box and *Add* the key to the selected list.
- Set the KeyboardHook property to "1 – This Task" when appropriate to enable the keyboard hook.
- Attach the following code to the WinHook control's KeyDownHook event:

[VB]

```
Private Sub KeyHook1_KeyDownHook(ByVal sender As Object, ByVal e As
AxDWSHK80Lib._DDwshkEvents_KeyDownHookEvent) Handles
KeyHook1.KeyDownHook
    e.discard = True
    SendKeys.Send("{TAB}")
End Sub
```

[C#]

```
private void KeyHook1_OnKeyDown(object sender,
AxDWSHK80Lib._DDwshkEvents_KeyDownHookEvent e)
{
    e.discard = true;
    SendKeys.Send("{TAB}");
}
```

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).

SpyWorksDotNet WinHook component implementation

- Add the Desaware.shcomponent.dll reference to your project. Declare a KeyHook object in the appropriate form, class or module. Create a new instance of the KeyHook object.

[VB]

```
Imports Desaware.SpyWorks
```

```
Friend KeyHook1 As KeyHook
KeyHook1 = New KeyHook()
```

```
[C#]
using Desaware.SpyWorks;
internal KeyHook KeyHook1;
KeyHook1 = new KeyHook();
```

2. Create a new instance of the KeyHook object's KeyList (KeyFilterList property) object.

```
[VB]
KeyHook1.Messages = New KeyFilterList()
```

```
[C#]
KeyHook1.Messages = new KeyFilterList();
```

3. Using the AddKey method of the KeyList object, add the FunctionKeys.Function_F2 and FunctionKeys.Function_F3 keys to the keys list.

```
[VB]
KeyHook1.KeyFilterList.AddKey(FunctionKeys.Function_F2, KeyFlags.Ctrl)
KeyHook1.KeyFilterList.AddKey(FunctionKeys.Function_F3, KeyFlags.Ctrl)
```

```
[C#]
KeyHook1.KeyFilterList.AddKey(FunctionKeys.Function_F2, KeyFlags.Ctrl);
KeyHook1.KeyFilterList.AddKey(FunctionKeys.Function_F3, KeyFlags.Ctrl);
```

4. Set the KeyHook object's Monitor property to "HookMonitor.EntireSystem".

```
[VB]
KeyHook1.Monitor = HookMonitor.EntireSystem
```

```
[C#]
KeyHook1.Monitor = HookMonitor.EntireSystem;
```

5. Set the KeyHook object's HookType property to "HookTypes.Keyboard".

```
[VB]
KeyHook1.HookType = HookTypes.Keyboard
```

```
[C#]
KeyHook1.HookType = HookTypes.Keyboard;
```

6. Create an OnKeyDown function to handle the KeyHook object's OnKeyDown event, and connect the event handler to the KeyHook object's OnKeyDown event. Add code to the OnKeyDown event to discard the Enter key and send the Tab key.

```
[VB]
AddHandler WinHook1.OnKeyDown, AddressOf WinHook1_OnKeyDown
```

```
Private Sub WinHook1_OnKeyDown (ByVal sender As Object, ByVal e As
KeyboardHookEventArgs)
    Select Case e.keycode
        Case FunctionKeys.Function_F2
            FunctionForCtrlF2()
        Case FunctionKeys.Function_F3
            FunctionForCtrlF3()
    End Select
```

```

        ' Next line is optional and depends on whether you want to disable
        ' the Ctrl+F2 & Ctrl+F3 hot keys that may affect other applications
        e.discard = True
End Sub

```

```

[C#]
KeyHook1.OnKeyDown += new KeyDownHookEventHandler(KeyHook1_OnKeyDown);

private void KeyHook1_OnKeyDown(object sender, KeyboardHookEventArgs e)
{
    switch (e.keycode)
    {
        case FunctionKeys.Function_F2:
            FunctionForCtrlF2();
            break;
        case FunctionKeys.Function_F3:
            FunctionForCtrlF2();
            break;
    }

    //Next line is optional and depends on whether you want to disable
    //the Ctrl+F2 & Ctrl+F3 hot keys that may affect other applications
    e.discard = true;
}

```

7. To start detecting for the message, set the KeyHook object's Enabled property to True.

```

[VB]
KeyHook1.Enabled = True

```

```

[C#]
KeyHook1.Enabled = true;

```

8. When finished using the KeyHook object, set the Enabled property to False to stop the message processing. Before deleting the object, disconnect the OnKeyDown event.

```

[VB]
KeyHook1.Enabled = False
RemoveHandler KeyHook1.OnKeyDown, AddressOf KeyHook1_OnKeyDown

```

```

[C#]
KeyHook1.Enabled = false;
KeyHook1.OnKeyDown -= new KeyDownHookEventHandler(KeyHook1_OnKeyDown);

```

dwshk80.ocx WinHook Control implementation

1. Add the Desaware SpyWorks Windows Hook Control component to your project. Add a WinHook control to Form1. To add a reference to the Hook control, select the Desaware dwshk80 v8 Hook Control from the COM tab of the Add Reference form. To add the Hook control to your form, you may need to add it to your toolbox first by using the Customize Toolbox command and adding the Desaware dwshk80 v8 Hook Control from the COM Components tab of the Customize Toolbox form.

2. Select the WinHook control's *Keys* property to display the Select Keys form. Select the F2 key from the Available Keys list box and check the Control check box, then *Add* the key to the selected list. Repeat the process for the F3 key.
3. Set the KeyboardHook property to "2 – Entire System" when appropriate to enable the keyboard hook.
4. Attach the following code to the WinHook control's KeyDownHook event:

```
[VB]
Private Sub WinHook1_KeyDownHook(ByVal sender As Object, ByVal e As
AxDWSHK80Lib._DDwshkEvents_KeyDownHookEvent) Handles
KeyHook1.KeyDownHook
    Select Case e.keycode
        Case Keys.F2
            FunctionForCtrlF2()
        Case Keys.F3
            FunctionForCtrlF3()
    End Select

    ' Next line is optional and depends on whether you want to disable
    ' the Ctrl+F2 & Ctrl+F3 hot keys that may affect other applications
    e.discard = True
End Sub

[C#]
private void WinHook1_KeyDownHook(object sender,
AxDWSHK80Lib._DDwshkEvents_KeyDownHookEvent e)
{
    switch (e.keycode)
    {
        case Keys.F2:
            FunctionForCtrlF2();
            break;
        case Keys.F3:
            FunctionForCtrlF2();
            break;
    }

    //Next line is optional and depends on whether you want to disable
    //the Ctrl+F2 & Ctrl+F3 hot keys that may affect other applications
    e.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 or online at <http://msdn.microsoft.com>.

Many SpyWorks examples demonstrate use of hooks.

Refer to the reference section for a complete list of properties and events for the Windows Hook control and the WinHook and KeyHook objects of the Desaware.shcomponent.dll component.

SpyWorks Concepts: dwshengine80.dll function library

IMPORTANT NOTE for those upgrading from previous versions of SpyWorks or Visual Basic: dwshengine80.dll is a replacement for dwspy36.dll (and before that dwspy32.dll) from earlier versions of SpyWorks. All functions in dwspy36.dll are included in dwshengine80.dll and are called in the same way. 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 or Visual Studio.

dwshengine80.dll function reference

The following functions are exported from dwshengine80.dll and are supported under Visual Studio .NET.

dwCopyData

This function can be used to copy arbitrary blocks of memory.

[VB]

```
Declare Sub dwCopyData Lib "dwshengine80.dll" (ByVal source As Integer,
ByVal dest As Integer, ByVal nCount As Integer)
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern void dwCopyData (int
source, int dest, int nCount);
```

source – Source address or object

dest – Destination address

nCount - The number of bytes to copy

Make sure that the destination is large enough to hold all of the data copied into it.

Under Win32 you can use the RtlMoveMemory API function. Note that the source and destination parameters for that function are opposite of this one.

NOTE: Although this function work in Visual Studio .NET, we recommend that you use the MarShal namespace instead. Refer to the SpyWorks .NET samples on using the MarShal namespace to copy data from pointers to .NET data types.

dwGetAddressForObject

This function returns a 32 bit address for the specified object type. You can overload the source parameter with different data types to retrieve the address for. In this example, it returns the address for an integer variable.

[VB]

```
Declare Function dwGetAddressForObject Lib "dwshengine80.dll" (source
As Integer) As Integer
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern int dwGetAddressForObject
(ref int source);
```

source – Source variable to return address for

NOTE: Although this functions work in Visual Studio .NET, we recommend that you use the Marshal namespace instead. Refer to the SpyWorks .NET samples on using the Marshal namespace to copy data from pointers to .NET data types.

dwXAllocateDataFrom

Allocates data from a foreign process's memory space. You can use the provided template dwSWXProcessClass class from the dwSpyCrossProcess file.

[VB]

```
Declare Function dwXAllocateDataFrom Lib "dwshengine80.dll" (ByVal size As Integer, ByVal foreignPID As Integer) As Integer
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern int dwXAllocateDataFrom (int size, int foreignPID);
```

size – Size of the buffer to allocate.

foreignPID – ID of the foreign process in which the data is allocated.

Call the dwXFreeDataFrom function to free the memory after you are done using it.

dwXFreeDataFrom

Free data from a foreign memory space which was allocated using dwXAllocateDataFrom. You can use the provided template dwSWXProcessClass class from the dwSpyCrossProcess file.

[VB]

```
Declare Sub dwXFreeDataFrom Lib "dwshengine80.dll" (ByVal foreignbuf As Integer, ByVal foreignPID As Integer)
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern void dwXFreeDataFrom (int foreignbuf, int foreignPID);
```

foreignbuf – Address of the foreign buffer to free.

foreignPID – ID of the foreign process in which the data is allocated.

dwXCopyAnsiStringFrom

Copies string data from a foreign process.

[VB]

```
Declare Function dwXCopyAnsiStringFrom Lib "dwshengine80.dll" (ByVal foreignbuf As Integer, ByVal foreignPID As Integer) As <MarshalAs(UnmanagedType.AnsiBStr)> String
```

[C#]

```
[DllImport("dwshengine80.dll")] [return: MarshalAs(UnmanagedType.AnsiBStr)] static extern string dwXCopyAnsiStringFrom (int foreignbuf, int foreignPID);
```

foreignbuf – Address of the string data.

foreignPID – ID of the foreign process.

The MarshalAs attribute is required so that the .NET Framework will free the BSTR.

Otherwise, it will result in a memory leak. You can use the provided template dwSWXProcessClass class from the dwSpyCrossProcess file.

dwXCopyUnicodeStringFrom

Copies string data from a foreign process.

[VB]

```
Declare Function dwXCopyUnicodeStringFrom Lib "dwshengine80.dll" (ByVal foreignbuf As Integer, ByVal foreignPID As Integer) As <MarshalAs(UnmanagedType.AnsiBStr)> String
```

[C#]

```
[DllImport("dwshengine80.dll")] [return: MarshalAs(UnmanagedType.AnsiBStr)] static extern string dwXCopyUnicodeStringFrom (int foreignbuf, int foreignPID);
```

foreignbuf – Address of the string data.

foreignPID – ID of the foreign process.

The MarshalAs attribute is required so that the .NET Framework will free the BSTR.

Otherwise, it will result in a memory leak. Note that this function Marshals the string as an Ansi string. This is because the function first converts the Unicode string into an Ansi BSTR before returning it. You can use the provided template dwSWXProcessClass class from the dwSpyCrossProcess file.

dwXCopyDataTo

Copies data from a local address to a foreign address.

[VB]

```
Declare Function dwXCopyDataTo Lib "dwshengine80.dll" (ByVal localdata As Integer, ByVal foreignbuf As Integer, ByVal size As Short, ByVal foreignPID As Integer) As Integer
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern int dwXCopyDataTo (int localdata, int foreignbuf, short size, int foreignPID);
```

localdata – Address of the local data.

foreign – Address to transfer the data to.

size – Size of the data to transfer.

foreignPID – ID of the foreign process.

You can overload this function with different data types for the *localdata* parameter depending on the data type involved. Refer to the provided template dwSWXProcessClass class from the dwSpyCrossProcess file for examples.

Returns -1 on error, 0 on success.

dwXCopyDataFrom

Copies data to a local address from a foreign address.

[VB]

```
Declare Function dwXCopyDataFrom Lib "dwshengine80.dll" (ByVal localdata As Integer, ByVal foreignbuf As Integer, ByVal size As Short, ByVal foreignPID As Integer) As Integer
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern int dwXCopyDataFrom (int localdata, int foreignbuf, short size, int foreignPID);
```

localdata – Local address to transfer the data to.

foreign – Foreign address to transfer the data from.

size – Size of the data to transfer.

foreignPID – ID of the foreign process.

You can overload this function with different data types for the *localdata* parameter depending on the data type involved. Refer to the provided template `dwSWXProcessClass` class from the `dwSpyCrossProcess` file for examples.

Returns -1 on error, 0 on success.

dwXGetModuleFileName

This function retrieves the executable file name (including path) for the specified process as long as the process has a window.

[VB]

```
Declare Function dwXGetModuleFileName Lib "dwshengine80.dll" (ByVal pid As Integer, ByVal modname As String) As String
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern string dwXGetModuleFileName (int pid, string modname);
```

pid – The ProcessID of the process to retrieve the file name for.

modname – The module name (the module name only, no extension or path). If the module name is NULL, the file name for the executable will be returned. Otherwise, the filename for the specified module will be returned.

You can use the provided template `dwSWXProcessClass` class from the `dwSpyCrossProcess` file.

dwXGetEditLine

Retrieves a line of text from a multi-line edit window located in another process.

[VB]

```
Declare Function dwXGetEditLine Lib "dwshengine80.dll" (ByVal hwnd As Integer, ByVal linenumber As Integer) As String
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern string dwXGetEditLine (int hwnd, int linenumber);
```

hwnd – Window handle of multi-line edit window to retrieve text from.

linenumber – Line number of the text to retrieve.

Returns the text of the specified line on success, empty string on error.

dwXSetForegroundWindow

Similar to the `SetForegroundWindow` API function except this function will force the specified window to be the foreground Window in Windows 2000 and XP (rather than flashing the Window's caption).

[VB]

```
Declare Function dwXSetForegroundWindow Lib "dwshengine80.dll" (ByVal hwnd As Integer) As Integer
```

[C#]

```
[DllImport("dwshengine80.dll")] static extern int  
dwXSetForegroundWindow (int hwnd);
```

hwnd – Window handle of window to bring to the foreground.

Returns -1 if invalid window handle, 0 if window cannot be brought to the foreground, > 0 on success.

Application Note: Using Cross Process Memory Access with SpyWorks

SpyWorks has long been established as the premier tool for cross task subclassing – allowing you to easily intercept messages from other applications. Once you have the message you can examine it and any data that it refers to. You can discard messages or modify its parameters.

But what if a message parameter contains a pointer to a string or structure in memory? Can you access that data?

Generally speaking the answer is no. Each Win32 process has its own memory space. That means that a pointer is only valid in the application that allocated the memory. Consider the problem of retrieving data from a rich text control.

Using the EM_GETTEXTRANGE message

The XTaskRichText sample application contains a single rich text control. A label control at the top of the form displays the window handle of the rich text control. It is loaded during the Form_Load event using the following code:

```
[VB]  
Private Sub frmGetText_Load(ByVal eventSender As System.Object, ByVal  
eventArgs As System.EventArgs) Handles MyBase.Load  
    Label1.Text = "Rich Text Hwnd: " + RichTextBox1.Handle.ToString  
End Sub  
  
[C#]  
private void frmGetText_Load(object sender, System.EventArgs e)  
{  
    label1.Text = "Rich Text Hwnd: " + richTextBox1.Handle.ToString();  
}
```

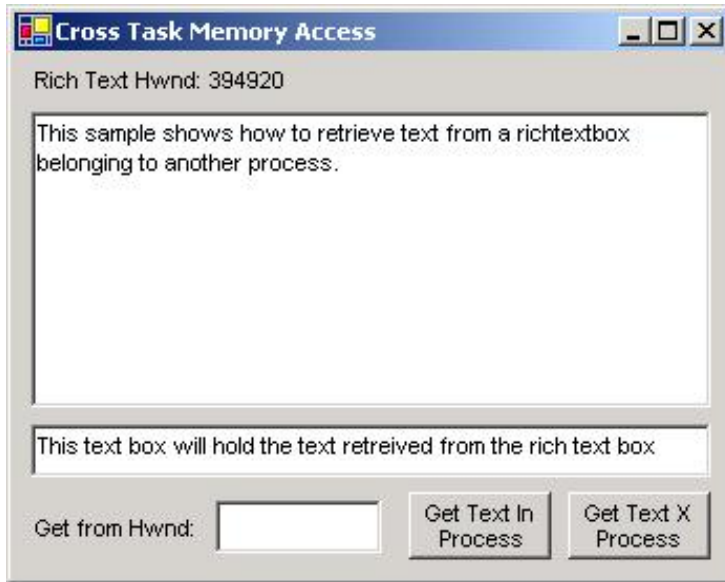


Figure 1 – XTaskRichText form in action.

The EM_GETTEXTRANGE message is used to retrieve text data from a rich text control. The wParam parameter of this message is always zero. The lParam parameter is a pointer to a TEXTRANGE structure that is defined as follows:

```
[VB]
<StructLayout(LayoutKind.Sequential, Pack:=1)> _
Friend Structure TEXTRANGE
    Dim chrg As CHARRANGE
    Dim lpstrText As Integer
End Structure

[C#]
[StructLayout(LayoutKind.Sequential, Pack=1)] public struct TEXTRANGE
{
    public CHARRANGE chrg;
    public int lpstrText;
}
```

The chrg field is a structure that defines the first and last character position of the text you want to retrieve from the rich text control. The lpstrText field is a pointer to a text buffer to load with the string data.

This structure contains a CHARRANGE structure that is defined as follows:

```
[VB]
<StructLayout(LayoutKind.Sequential, Pack:=1)> _
Friend Structure CHARRANGE
    Dim cpMin As Integer
    Dim cpMax As Integer
End Structure

[C#]
[StructLayout(LayoutKind.Sequential, Pack=1)] public struct CHARRANGE
{
```

```

        public int cpMin;
        public int cpMax;
    }

```

Let's take a look at how you would use this message to retrieve text from rich text box that exists within your own process:

In Process Example

The following code retrieves text from a rich text box within the current process.

```

[VB]
Private Declare Auto Function SendMessageEXTRANGE Lib "user32" Alias
"SendMessage" (ByVal hwnd As Integer, ByVal wParam As Integer, ByVal
wParam As Integer, ByVal lParam As EXTRANGE) As Integer

Private Const WM_USER As Integer = &H400
Private Const EM_GETEXTRANGE As Integer = WM_USER + 75

Private Sub cmdGet_Click(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles cmdGet.Click
    Dim tr As EXTRANGE
    Dim deststring As String
    Dim res As Integer
    Dim useHwnd As Integer

    If txtHwnd.Text = "" Then
        MessageBox.Show("Invalid window")
        Exit Sub
    End If
    useHwnd = CInt(txtHwnd.Text)
    tr.chrg.cpMax = 1023
    tr.lpstrText = Marshal.AllocHGlobal(1024).ToInt32

    res = SendMessageEXTRANGE(useHwnd, EM_GETEXTRANGE, 0, tr)
    If res > 0 Then
        deststring =
Marshal.PtrToStringAuto(IntPtr.op_Explicit(tr.lpstrText))
        Text1.Text = deststring
    Else
        MessageBox.Show("EM_GETEXTRANGE failed")
    End If

    Marshal.FreeHGlobal(IntPtr.op_Explicit(tr.lpstrText))
End Sub

[C#]
[DllImport("user32.dll", CharSet= CharSet.Auto)] private static extern
int SendMessage (int hwnd, int wParam, int lParam, ref EXTRANGE lParam);

private const int WM_USER = 0x400;
private const int EM_GETEXTRANGE = WM_USER + 75;

private void cmdGet_Click(object sender, System.EventArgs e)
{
    EXTRANGE tr;

```

```

string deststring;
int res, useHwnd;

if (txtHwnd.Text == "")
{
    MessageBox.Show("Invalid window");
    return;
}
useHwnd = Convert.ToInt32(txtHwnd.Text);
tr = new TEXTRANGE();
tr.chrg.cpMax = 1023;
tr.lpstrText = Marshal.AllocHGlobal(1024).ToInt32();

res = SendMessage(useHwnd, EM_GETTEXTRANGE, 0, ref tr);
if (res > 0)
{
    deststring = Marshal.PtrToStringAuto((IntPtr)tr.lpstrText);
    textBox1.Text = deststring;
}
else
    MessageBox.Show("EM_GETTEXTRANGE failed");

Marshal.FreeHGlobal((IntPtr)tr.lpstrText);
}

```

Let's take a closer look at this code.

First, you should enter the window handle of the rich text box into the txtHwnd.Text control. Type in the same number that you see in the label control. This window handle is copied into the useHwnd variable.

```

[VB]
useHwnd = CInt(txtHwnd.Text)
[C#]
useHwnd = Convert.ToInt32(txtHwnd.Text);

```

For the purposes of this example, we arbitrarily load only the first 1023 characters. A real example would use other messages to determine the length of available text. The EM_GETTEXTRANGE considers the cpMax field of the CHRANGE structure to represent the maximum character number, so setting cpMin to zero and cpMax to 1023 will retrieve the entire contents of the rich text box up to 1023 characters.

```

[VB]
tr.chrg.cpMax = 1023
[C#]
tr.chrg.cpMax = 1023;

```

The lpstrText field of the TEXTRANGE structure needs to be a pointer to a buffer to load with the text. In this case the buffer will be a memory buffer returned by Marshal.AllocHGlobal.

```

[VB]
tr.lpstrText = Marshal.AllocHGlobal(1024).ToInt32
[C#]

```

```
tr.lpstrText = Marshal.AllocHGlobal(1024).ToInt32();
```

Next the `SendMessageTEXTRANGE` (alias of `SendMessage`) function sends the message to the rich text control, passing a pointer (by reference) to the `TEXTRANGE` structure as the `IParam` parameter as follows:

```
[VB]
res = SendMessageTEXTRANGE(useHwnd, EM_GETTEXTRANGE, 0, tr)
[C#]
res = SendMessage(useHwnd, EM_GETTEXTRANGE, 0, ref tr);
```

The message returns the number of characters read. If non zero, the function uses the `Marshal.PtrToStringAuto` function to copy the data from the buffer into a string.

```
[VB]
If res > 0 Then
    deststring =
Marshal.PtrToStringAuto(IntPtr.op_Explicit(tr.lpstrText))
    Text1.Text = deststring
Else
    MessageBox.Show("EM_GETTEXTRANGE failed")
End If
[C#]
if (res > 0)
{
    deststring = Marshal.PtrToStringAuto((IntPtr)tr.lpstrText);
    textBox1.Text = deststring;
}
else
    MessageBox.Show("EM_GETTEXTRANGE failed");
```

Finally, the memory buffer is freed by calling `Marshal.FreeHGlobal`.

```
[VB]
Marshal.FreeHGlobal(IntPtr.op_Explicit(tr.lpstrText))
[C#]
Marshal.FreeHGlobal((IntPtr)tr.lpstrText);
```

Cross Process Example

Try running a second instance of the `XTaskGet` program. Enter the window handle of the second application's rich text box into the window handle text box of the first application, and click on the "Get Text InProc" button.

You'll get a message indicating that the `EM_GETTEXTRANGE` failed.

Why? Because the `IParam` parameter of the `EM_GETTEXTRANGE` message was set to point to the location in memory of the `TEXTRANGE` structure in your application. But this memory location is invalid in the other process. You are sure to get either a memory exception or some sort of memory corruption when the other process tries to read the invalid memory location or load data into the string buffer (which is also a pointer to an invalid memory location).

In order to solve this problem, you must find a way to allocate a memory buffer in the second application (which will be referred to from now on as the "foreign" application, to distinguish it from the running application that is sending the message). You'll also need a way to copy data to and from this foreign application. Fortunately, Desaware's SpyWorks makes it easy to handle both operations. The cmdGetX_Click event is shown below:

```
[VB]
Private Sub cmdGetX_Click(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles cmdGetX.Click
    Dim tr As TEXTRANGE
    Dim ForeignProcessId As Integer
    Dim deststring As String
    Dim res As Integer
    Dim useHwnd As Integer
    Dim textstringlen As Integer

    If txtHwnd.Text = "" Then
        MessageBox.Show("Invalid window")
        Exit Sub
    End If

    useHwnd = CInt(txtHwnd.Text)
    GetWindowThreadProcessId(useHwnd, ForeignProcessId)

    Dim bytearrayXProcess As New dwSWXProcessClass(ForeignProcessId,
1024)

    tr.chrg.cpMax = 1023
    tr.lpstrText = bytearrayXProcess.XProcessBuffer

    Dim textrangeXProcess As New dwSWXProcessClass(ForeignProcessId,
CType(Marshal.SizeOf(tr), Short))

    textrangeXProcess.CopyToMemoryByTEXTRANGE(tr)

    res = SendMessage(useHwnd, EM_GETTEXTRANGE, 0,
textrangeXProcess.XProcessBuffer)

    If res > 0 Then
        textrangeXProcess.CopyFromMemoryByTEXTRANGE(tr)

        deststring = bytearrayXProcess.GetAnsiStringFromMemory()
        textstringlen = deststring.Length

        ' if the string length does not match, then try unicode string
        If textstringlen <> res Then
            deststring = bytearrayXProcess.GetUnicodeStringFromMemory()
        End If

        Text1.Text = deststring

    Else
        MessageBox.Show("EM_GETTEXTRANGE failed.")
    End If
End Sub
```



```

        textrangeXProcess.FreeMemory()
        bytearrayXProcess.FreeMemory()
End Sub

[C#]
private void cmdGetX_Click(object sender, System.EventArgs e)
{
    TEXTRANGE tr = new TEXTRANGE();
    int ForeignProcessId = 0;
    string deststring;
    int res;
    int useHwnd;
    int textstringlen;

    if (txtHwnd.Text == "")
    {
        MessageBox.Show("Invalid Window");
        return;
    }

    useHwnd = Convert.ToInt32(txtHwnd.Text);

    GetWindowThreadProcessId(useHwnd, ref ForeignProcessId);

    dwSWXProcessClass bytearrayXProcess = new
dwSWXProcessClass(ForeignProcessId, 1024);

    tr.chrg.cpMax = 1023;
    tr.lpstrText = bytearrayXProcess.XProcessBuffer;

    dwSWXProcessClass textrangeXProcess = new
dwSWXProcessClass(ForeignProcessId, (short)Marshal.SizeOf(tr));

    textrangeXProcess.CopyToMemoryByTEXTRANGE(ref tr);

    res = SendMessage(useHwnd, EM_GETTEXT, 0,
textrangeXProcess.XProcessBuffer);

    if (res > 0)
    {
        textrangeXProcess.CopyFromMemoryByTEXTRANGE(ref tr);

        deststring = bytearrayXProcess.GetAnsiStringFromMemory();
        textstringlen = deststring.Length;

        // if the string length does not match, then try unicode string
        if (textstringlen != res)
            deststring =
bytearrayXProcess.GetUnicodeStringFromMemory();

        textBox1.Text = deststring;
    }
    else
        MessageBox.Show("EM_GETTEXT failed.");
}

```

```

        textrangeXProcess.FreeMemory();
        bytearrayXProcess.FreeMemory();
    }

```

Let's look at this function in detail.

Several new variables are used. The ForeignProcessId is used to hold the process ID of the process containing the rich text box.

```

[VB]
Dim ForeignProcessId As Integer
[C#]
int ForeignProcessId = 0;

```

The bytearrayXProcess and textrangeXProcess variables contain the new dwSWXProcessClass object defined in the dwSpyWorksCrossProcess module file. The dwSWXProcessClass class wraps some of the SpyWorks Cross Process functions and is used to represent a block of memory in another process.

The bytearrayXProcess variable allocates a block of memory in the foreign process and will contain the text retrieved from the rich text box.

```

[VB]
Dim bytearrayXProcess As New dwSWXProcessClass(ForeignProcessId, 1024)
[C#]
dwSWXProcessClass bytearrayXProcess = new
dwSWXProcessClass(ForeignProcessId, 1024);

```

The textrangeXProcess variable allocates a block of memory in the foreign process and will contain a TEXTRANGE structure.

```

[VB]
Dim textrangeXProcess As New dwSWXProcessClass(ForeignProcessId,
CType(Marshal.SizeOf(tr), Short))
[C#]
dwSWXProcessClass textrangeXProcess = new
dwSWXProcessClass(ForeignProcessId, (short)Marshal.SizeOf(tr));

```

The useHwnd variable and TEXTRANGE cpMax field is set in the same way as in the previous application. The only difference is that the txtHwnd text box should contain the window handle of the rich text box from the foreign application.

```

[VB]
If txtHwnd.Text = "" Then
    MessageBox.Show("Invalid window")
    Exit Sub
End If
useHwnd = CInt(txtHwnd.Text)
tr.chrg.cpMax = 1023

[C#]
if (txtHwnd.Text == "")
{
    MessageBox.Show("Invalid window");
    return;
}

```

```

}
useHwnd = Convert.ToInt32(txtHwnd.Text);
tr = new TEXTRANGE();
tr.chrg.cpMax = 1023;

```

It's easy to obtain the process ID of the foreign process. The `GetWindowThreadProcessId` function retrieves the thread ID and process ID for any window. In this case you'll use the window handle of the foreign rich text box as shown here:

```

[VB]
Private Declare Function GetWindowThreadProcessId Lib "user32" (ByVal
hwnd As Integer, ByRef lpdwProcessId As Integer) As Integer
GetWindowThreadProcessId(useHwnd, ForeignProcessId)

```

```

[C#]
[DllImport("user32.dll")] private static extern int
GetWindowThreadProcessId (int hwnd, ref int lpdwProcessId);
GetWindowThreadProcessId(useHwnd, ref ForeignProcessId);

```

We need two blocks of memory in the foreign process - one 1024 characters long to hold the text copied from the rich text box. Another needs to be the size of a `TEXTRANGE` structure to hold the retrieval information. The `bytearrayXProcess` and `textrangeXProcess` objects were earlier created with the required memory blocks.

You can see that we still have a local `TEXTRANGE` variable named `tr` which is loaded with data in a similar manner as before. The difference is that the `lpstrText` field now refers to the `bytearrayXProcess` variable's `XProcessBuffer` property that contains a memory address in another process to hold the string data. We'll copy the local `TEXTRANGE` structure into the foreign memory buffer referred to by the `textrangeXProcess` variable, then pass the memory address of the `textrangeXProcess` buffer with the `EM_GETTEXTRANGE` message.

```

[VB]
tr.lpstrText = bytearrayXProcess.XprocessBuffer
textrangeXProcess.CopyToMemoryByTEXTRANGE(tr)
res = SendMessage(useHwnd, EM_GETTEXTRANGE, 0,
textrangeXProcess.XProcessBuffer)

```

```

[C#]
tr.lpstrText = bytearrayXProcess.XProcessBuffer;
textrangeXProcess.CopyToMemoryByTEXTRANGE(ref tr);
res = SendMessage(useHwnd, EM_GETTEXTRANGE, 0,
textrangeXProcess.XProcessBuffer);

```

The return value from sending the `EM_GETTEXTRANGE` contains the length of the string retrieved. If a string was retrieved successfully, we retrieve the string from the foreign buffer.

Now in this particular case, we don't really need to copy the `TEXTRANGE` structure from the foreign memory block back to the local structure, because the foreign process didn't change it during the message processing. The `textrangeXProcess` variable's `CopyFromMemoryByTEXTRANGE` function can be used to copy the data back.

```
[VB]
textrangeXProcess.CopyFromMemoryByTEXTRANGE(tr)
[C#]
textrangeXProcess.CopyFromMemoryByTEXTRANGE(ref tr);
```

If a string was retrieved successfully, we call the `bytearrayXProcess` variable's `GetAnsiStringFromMemory` or `GetUnicodeStringFromMemory` function to retrieve the string and set the text box. Now all we need to figure out is whether an ansi or unicode string was returned in the buffer. But here's an issue we ran into. When the process id is a .NET application containing the .NET edition of the `RichTextBox` control, it places a unicode string in the buffer. The same test done on a VB 6 compiled application containing the VB6 edition of the `RichTextBox` control returns an ansi string. It doesn't matter how `SendMessage` was declared, we would get the same behavior. Since `SendMessage` returns the length of the string, we know how long the data is supposed to be. So, we try ansi string first, and then if the string length does not match, we try the unicode string.

```
[VB]
deststring = bytearrayXProcess.GetAnsiStringFromMemory()
textstringlen = deststring.Length

' if the string length does not match, then try unicode string
If textstringlen <> res Then
    deststring = bytearrayXProcess.GetUnicodeStringFromMemory()
End If

Text1.Text = deststring

[C#]
deststring = bytearrayXProcess.GetAnsiStringFromMemory();
textstringlen = deststring.Length;

// if the string length does not match, then try unicode string
if (textstringlen != res)
    deststring = bytearrayXProcess.GetUnicodeStringFromMemory();

textBox1.Text = deststring;
```

Finally, it is important to free the memory blocks so that you avoid creating a memory leak in the foreign application:

```
[VB]
textrangeXProcess.FreeMemory()
bytearrayXProcess.FreeMemory()
[C#]
textrangeXProcess.FreeMemory();
bytearrayXProcess.FreeMemory();
```

Conclusion

Working with foreign memory spaces can be an incredibly complex task, as attested to the fact that a number of very advanced articles have been written on the subject for C++

programmers. However, Desaware's SpyWorks includes a robust set of functions that make working with foreign memory buffers almost as easy as working with local memory. These functions are wrapped by the dwSWXProcessClass object contained in the dwSpyWorksCrossProcess module file.

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. SpyWorks 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. Functionality remains unchanged for SpyWorks 8 except that only the 1.1 framework is currently supported (support for the 2.0 framework will be available on release).

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 or from C# using the DllImport attribute. In order for a function to be accessible using the Declare statement or DllImport attribute, it must be "exported" - this means that the name of the function and its location in the DLL is made public by the DLL.

.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 .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 previous versions of Visual Basic (in which case you use the Declare statement), or written in other languages.
- 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.
- Some 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 Studio .NET DLL is a two-step process.

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

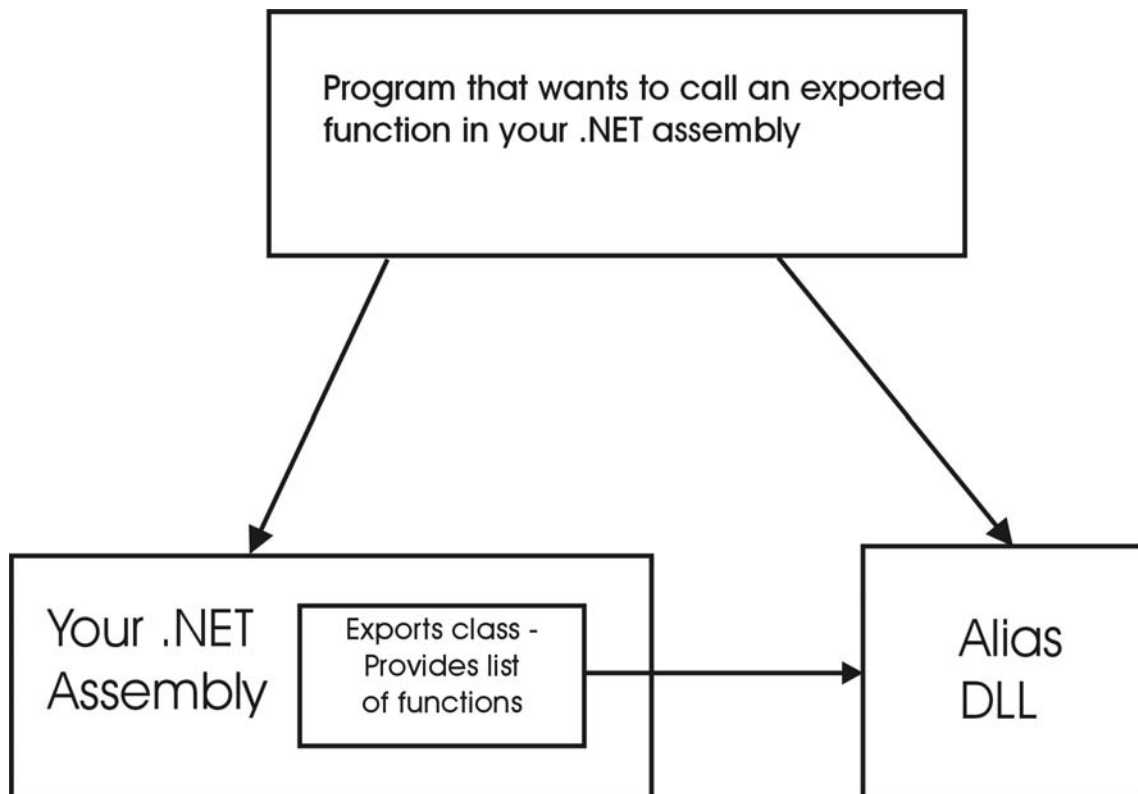
But first, we recommend you read a little bit about how Desaware's Dynamic Export Technology Works.

How Dynamic Export Technology Works

Desaware's Dynamic Export Technology works without modifying your .NET assembly in any way. This has two advantages:

- It eliminates any chance that Desaware's tools might corrupt your assembly 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.



First, you'll define a Public class called Exports in your .NET project. This class has a number of methods that will be called by SpyWorks when your assembly loads in order to find out about the functions that it wants to export.

The real work is done by a separate DLL called an Alias DLL. 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. It then creates the Alias DLL with the file name that you specify.

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 library 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 assembly DLL and read the Exported attribute information to obtain the list of

functions that you wish to export. It will then link those functions in dynamically to be accessed by the calling application.

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

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.

The Exports Class

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 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, and then 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).

[VB]

```
<Exported("MyExportFunc", 1)> Public Shared Function MyExportFunc  
(ByVal x As Integer) As Integer  
<Exported("MyExport", 2, CCall:=True)> Public Shared Function MyExport  
(ByVal x As Integer) As Integer
```

[C#]

```
[Exported("MyExportFunc", 1)] public static int MyExportFunc (int x)  
[Exported("MyExport", 2, Ccall=True)] public static int MyExport (int  
x)
```

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 .NET project. You do not need to create a new alias DLL when you make changes to the functions being exported.

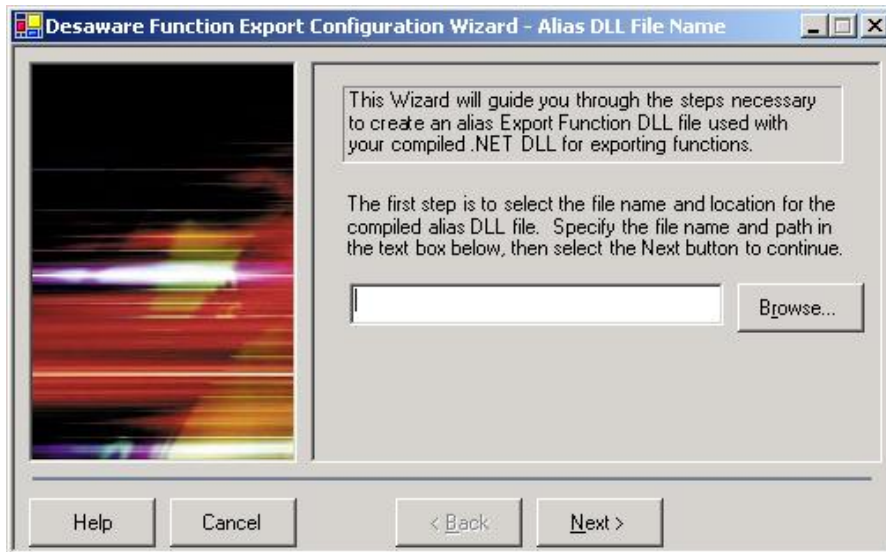
Currently, byval and byref passing of the following function parameter and return data types are supported:

- 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 ExportWizard

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



Alias DLL File Name Step: This step is used to specify the path and file name of your alias DLL.

DLL and Assembly Name Step: 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 Step: 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 Step: This step scans the specified Assembly for the required declarations, formatting errors, and other inconsistencies regarding the Assembly.

Compile Step: 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 assembly must be running within the same process space as the calling function.

The Visual Studio IDE allows you to attach an external application to test your .NET assemblies. Open your Export Function project and compile it in Debug mode. Then set the Project's Configuration Debugging properties such that it starts an external program that will call the exported functions from your Export Function assembly. Set breakpoints in your Export Function project's code and run the project under the Visual Studio IDE.

Distributing your Exported Function files

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.

Warning! Exporting Functions is Dangerous!

When you export a function from a DLL assembly 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 .NET - 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.

.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 would 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.

Migrating to the Desaware.shcomponent.dll

Introduction

The Desaware.shcomponent.dll component is a native .NET component that exposes Windows Subclass and Windows Hooks technology previously found in the SpyWorks

Subclass control and SpyWorks WinHook control. Desaware recommends using the new Desaware.shcomponent.dll component in place of the previous Subclass and WinHook COM controls for development on the Visual Studio .NET platform. In order to migrate to the new component, you will need to make some code changes. Some of these are due to differences in functionality between the components or simple “cleanup” of properties and methods – changes not possible in the COM controls due to the need to preserve backwards compatibility. Other changes are more fundamental – changes in approach that result from the huge differences between .NET and COM. This section will walk you through the changes necessary in order to migrate the Subclass or WinHook controls to the new component, both in terms of code and understanding the key concepts. Note: To migrate from the SpyWorks 7.1 Desaware.SpyWorksDotNet.Dll component to the new Desaware.shcomponent.dll component, all you need to do is remove the reference to one and add a reference to the other. They are functionally equivalent. Desaware.shcomponent.dll uses the newer dwshengine80.dll subclassing/hook engine.

Fundamental Differences between the Desaware.shcomponent.dll component and the COM based subclassing and hook components.

To understand the differences between the COM and .NET components, you need to remember that the COM controls don’t actually do any subclassing or hooking. Instead, they work with the dwshengine80.dll subclassing engine.

The dwshengine80.dll is responsible for all actual subclassing and hooking in SpyWorks. It handles all interprocess communication, and does low level filtering to maximize performance. More important – by placing these lower level functions in a DLL that has minimal dependencies, the potential impact of SpyWorks on a system is dramatically reduced. Instead of trying to map the entire VB6 or .NET runtime into another process (which happens during hooks or cross task subclassing), only the dwshengine80.dll DLL is mapped.

The dwshengine80.dll component is a native API DLL – it has no runtime and doesn’t even use COM internally.

The dwshengine80.dll component replaces the dwspy36.dll component used in SpyWorks 7.1. It incorporates a more stable hook system that is better able to recover after application crashes (

The subclassing and hook controls use the dwshengine80 DLL to perform their tasks. They provide the design time interface and an easy to use programmatic interface (properties, methods, events) to work with. More important – they are responsible for guaranteeing that the COM threading rules required by VB6 are followed – that all events are raised on the thread that belongs to the form on which the control was placed. The controls also handle posted events – the ability to process events asynchronously by posting them to the control’s message queue and handling them at a later (safe) time.

The Desaware.shcomponent.dll component

The Desaware.shcomponent.dll component also uses the dwshengine80.dll engine. This is important, because this preserves all the advantages of this component – the light

footprint, the efficiency, and the fact that it is a reliable component, proven on millions of systems worldwide.

However, the entire approach for the client component changes in .NET, because all of the things that an ActiveX control must do to allow compatibility with VB6, are not necessary in VB .NET or C#.

The .NET languages are free threaded – thus there is no longer any need to marshal all events to a particular thread (which has a performance cost). This means that you no longer need a control at all (since the main reason for using a control was to have a window to use to implement thread marshaling). Without a window handle, you can't use message posting for asynchronous event handling, but that's ok because .NET has native asynchronous delegates.

In other words, the Desaware.shcomponent.dll component is designed from scratch based on the principles of sound .NET programming. It is not an adaptation from the COM controls.

The key fundamental differences between the COM controls and the .NET component are as follows:

1. The Desaware.shcomponent.dll component is implemented as a class library rather than a control. As such, there is no design time behavior – you simply create an instance of the class and set the properties and methods in your code.
2. Because the Desaware.SpyWorksDotNet component is a class library, it no longer requires a host form. This allows it to be used from other application types. Note, however, that in order to function properly the process that is using the component must have at least one thread that is running a message pump. You can use the Application object (Application.Run) to create a message pump on a thread even for non-form based applications such as console application.
3. Events in this control can be raised on any thread. So if you are accessing a form property, you must use the form's Invoke method to be sure the property is accessed on the correct thread. You should also be sure to do thread synchronization where necessary.
4. When handling messages asynchronously, this component uses asynchronous delegates – this means that events will be raised on a different thread from the original message.

While the thread management in some cases may require additional code (compared to the COM controls), the result is a much more lightweight and efficient component.

Major Changes to the Subclassing Component

In addition to the differences listed previously that apply to all of the SpyWorks components (subclassing, hooks and keyboard hooks), the following significant differences apply to the Subclasser component.

- Each Desaware.SpyWorks.Subclasser object can subclass only one window. The COM based control was designed to subclass multiple windows because each control carried a very high overhead in terms of system resources (including all of the ActiveX control overhead and a window handle). Allowing each control to subclass multiple windows allowed all of the subclassed windows to share those

- resources. The new Subclasser object is a native .NET class with negligible overhead, so there is no harm in using separate objects to subclass each window.
- In following the common .NET design pattern, messages to be subclassed are specified using a WindowsMessageList collection object that is accessed via the Messages property.
 - Events use the common .NET design pattern of two parameters for each event: one the sender, the other an object that derives from EventArgs

Major Changes to the Windows hook Component

In addition to the differences listed previously that apply to all of the SpyWorks components (subclassing, hooks and keyboard hooks), the following significant differences apply to the WinHook component.

- In following the common .NET design pattern, messages to be detected are specified using a WindowsMessageList collection object that is accessed via the Messages property.
- Two new hook types: WH_FOREGROUNDIDLE and WH_MOUSE_LL are supported with this component.
- Events use the common .NET design pattern of two parameters for each event: one the sender, the other an object that derives from EventArgs
- The Monitor property can no longer specify the ThisForm or MySiblings options because the component is no longer associated with a form.

Major Changes to the Keyboard hook Component

The Desaware.SpyWorks.Keyhook class derives from the Desaware.SpyWorks.Winhook class, incorporating additional properties and methods specific to keyboard handling.

In addition to the differences listed previously that apply to all of the SpyWorks components (subclassing, hooks and keyboard hooks), the following significant differences apply to the KeyHook component.

- There is no kbdhook event. The OnKeyDown and OnKeyUp events are separate.
- Both regular (WH_KEYBOARD) and low level (WH_KEYBOARD_LL) hooks are supported.
- In following the common .NET design pattern, keys to be detected are specified using a KeyList collection object that is accessed via the KeyFilterList property.
- Events use the common .NET design pattern of two parameters for each event: one the sender, the other an object that derives from EventArgs.
- With SpyWorks 8, character keystrokes destined to browser windows and text boxes with the password style are not detected for processes other than your own. This is part of the new anti-spyware feature of SpyWorks 8.

Migrating the Subclass control from a .NET project

1. Write down the design time properties for each of the subclass controls, especially messages that the control is detecting.
2. Delete all subclass controls from all forms - you may also need to remove a line of code from the form the control is on similar to:

```
[VB]
Friend WithEvents SubClass1 As
AxDesaware.SpyWorks.dwsbcNET.AxSubClass
```

```
[C#]
private AxDesaware.SpyWorks.dwsbcNET.AxSubClass SubClass1;
```

The library may be AxDWSBC80Lib if you are migrating from the SpyWorks 8 ActiveX controls.

3. Remove the project reference to Desaware.SpyWorks.dwsbcNet or DWSBC80LIB and AxDWSBC80LIB (or DWSBC36LIB if migrating from SpyWorks 7.1 or earlier).
4. Add a project reference to the new Desaware.shcomponent.dll .NET component.
5. Declare the new Subclass object. In VB .NET, you can use the WithEvents declaration instead of adding events explicitly.

```
[VB]
Friend SubClass1 As Desaware.SpyWorks.Subclasser
[C#]
internal Desaware.SpyWorks.Subclasser SubClass1;
```

6. Add code to create a new instance of the object.

```
[VB]
SubClass1 = New Desaware.SpyWorks.Subclasser()
[C#]
SubClass1 = new Desaware.SpyWorks.Subclasser();
```

7. Write code to restore the previous Subclass control's design time properties after creating the Subclasser object. Usually, you would just set the Messages, HwndParam and Type properties for the Subclass control. All three of these properties have been changed for the new Subclasser object. Refer to the *Property changes* section for more information.

```
[VB]
Imports Desaware.SpyWorks

SubClass1.Messages = New WindowsMessageList()
SubClass1.Messages.AddMessage(StandardMessages.WM_ACTIVATE)
SubClass1.Messages.AddMessage(StandardMessages.WM_INITMENUPOPUP)

SubClass1.SubclassingType = SubclassingTypes.PostDefault
```

```

AddHandler SubClass1.OnWndMessage, AddressOf SubClass1_OnWndMessage

SubClass1.HwndParam = IntPtr.op_Explicit(hwnd)

[C#]
using Desaware.SpyWorks;

SubClass1.Messages = new WindowsMessageList();
SubClass1.Messages.AddMessage(StandardMessages.WM_ACTIVATE);
SubClass1.Messages.AddMessage(StandardMessages.WM_INITMENUPOPUP);

SubClass1.SubclassingType = SubclassingTypes.PostDefault;
SubClass1.OnWndMessage += new
WndMessageEventHandler(SubClass1_OnWndMessage);

SubClass1.HwndParam = (IntPtr)hwnd;

```

8. Replace your Subclass control's WndMessage or WndMessageX event with the OnWndMessage event and move your code to the new event. The OnWndMessage event is declared as follows:

```

[VB]
Private Sub SubClass1_OnWndMessage(ByVal sender As Object, ByVal e As
Desaware.SpyWorks.WndMessageEventArgs)
[C#]
private void SubClass1_OnWndMessage(object sender,
Desaware.SpyWorks.WndMessageEventArgs e)

```

9. The WndMessageEventArgs data type found in the OnWndMessage event is similar to the AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageEvent and AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageXEvent data types. The exception is the WndMessageEventArgs.hwnd field that is now an IntPtr instead of an Integer. The WndMessageEventArgs.process field is set to zero if the Windows message subclassed is from the process containing the subclasser object. Otherwise, it contains the process id of the process the subclassed Windows message was intended for.

10. Remove the OnWndMessage event handler when done using the subclass object.

```

[VB]
RemoveHandler SubClass1.OnWndMessage, AddressOf SubClass1_OnWndMessage
[C#]
SubClass1.OnWndMessage -= new
WndMessageEventHandler(SubClass1_OnWndMessage);

```

Property changes:

AddHwnd	Obsolete. Each instance of the Subclasser object can only subclass a single window.
---------	---

ClearMessage	Obsolete. Use the Messages.RemoveMessage method to remove a Windows message from the list of messages being subclassed.
CtlParam	Obsolete. Use the HwndParam property to assign the window to subclass at run time.
HookCount	Obsolete. Each instance of the Subclasser object can only subclass a single window.
HookEnabled	Replaced by the Enabled property.
HwndArray	Obsolete. Each instance of the Subclasser object can only subclass a single window.
HwndParam	Changed from an Integer data type to an IntPtr data type. You can use the IntPtr.op_Explicit function to cast an Integer data type to an IntPtr data type.
MessageArray	Obsolete. Use the Messages.Messages property to retrieve an array of the messages being subclassed.
MessageCount	Obsolete. Use the Messages.Messages.Length property to retrieve the number of messages being subclassed.
Messages	The data type for this property has changed. It use to be such that you were able to assign a messages by setting the Messages in design time or at run time by setting the Messages property to a message number. Now Messages refer to a WindowsMessageList data type (an arraylist). This property is initially set to Nothing which will subclass all Windows messages. Most Windows messages are also exposed through one of the Desaware.SpyWorks.*Messages enumerator.
Persist	Obsolete.
PostEvent	Obsolete. Use an asynchronous delegate if you need to defer an operation.
PostOnFreeze	Obsolete. Events are never “frozen”
PostOnFreezeMax	Obsolete.
RegMessage	Obsolete. Use the GetRegisteredWindowMessage property to retrieve a unique Windows message number on the current

	system for the specified string. Then use the <code>Messages.AddMessage</code> to include the message for subclassing.
<code>RegMessageNum</code>	Obsolete. Use the <code>GetRegisteredWindowMessage</code> property to retrieve a unique Windows message number on the current system for the specified string.
<code>RemoveHwnd</code>	Obsolete. Each instance of the <code>Subclasser</code> object can only subclass a single window.
<code>StyleChangeArray</code>	Obsolete.
<code>Type</code>	Replaced by the <code>SubclassingType</code> , set this to a value exposed by the <code>Desaware.SpyWorks.SubclassingTypes</code> enumerator.
<code>UseDirectInterface</code>	Obsolete.
<code>UseOnlyXEvent</code>	Obsolete.

Method changes:

<code>GetAnsiString</code>	The <i>address</i> parameter is now an <code>IntPtr</code> data type instead of an <code>Integer</code> data type.
<code>GetUnicodeString</code>	The <i>address</i> parameter is now an <code>IntPtr</code> data type instead of an <code>Integer</code> data type.
<code>StyleChangeOn</code>	Obsolete.

Event changes:

<code>WndMessage</code>	Replaced by the <code>OnWndMessage</code> event. This event's <code>AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageEvent</code> type is replaced by the <code>WndMessageEventArgs</code> type that exposes the same fields. The exception is the <i>hwnd</i> field that is now an <code>IntPtr</code> type instead of an <code>Integer</code> type.
<code>WndMessageX</code>	Replaced by the <code>OnWndMessage</code> event. This event's <code>AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageXEvent</code> type is replaced by the <code>WndMessageEventArgs</code> type that exposes the same fields. The exception is the <i>hwnd</i> field that is now an <code>IntPtr</code> type instead of

	an Integer type.
--	------------------

Migrating the WinHook control from a .NET project – Windows hook migration

1. Write down the design time properties for each of the winhook controls, especially keys that the control is detecting.
2. Delete all winhook controls from all forms - you may also need to remove a line of code from the form the control is on similar to:

```
[VB]
Friend WithEvents WinHook1 As AxDesaware.SpyWorks.dwshkNET.AxWinHook
```

```
[C#]
private AxDesaware.SpyWorks.dwshkNET.AxWinHook WinHook1;
```

The library may be AxDWSHK80Lib if you are migrating from the SpyWorks 8 ActiveX controls.

3. Remove the project reference to Desaware.SpyWorks.dwshkNet or DWSHK80LIB and AxDWSHK80LIB (or DWSBC36LIB if migrating from SpyWorks 7.1 or earlier).
4. Add a project reference to the new Desaware.shcomponent.dll .NET component.
5. Declare the new WinHook object as follows. In VB .NET, you can use the WithEvents declaration instead of adding events explicitly.

```
[VB]
Friend WinHook1 As Desaware.SpyWorks.WinHook
```

```
[C#]
internal Desaware.SpyWorks.WinHook WinHook1;
```

6. Add code to create a new instance of the object.

```
[VB]
WinHook1 = New Desaware.SpyWorks.WinHook()
```

```
[C#]
WinHook1 = new Desaware.SpyWorks.WinHook();
```

7. Write code to restore the previous WinHook control's design time properties after creating the WinHook object. Usually, you would just set the Messages, Monitor, HookType and HookEnabled properties for the WinHook control. Some of these properties have changed for the new WinHook object. Refer to the *Property changes* section for more information.

```

[VB]
Imports Desaware.SpyWorks

WinHook1.Messages = New WindowsMessageList()
WinHook1.Messages.AddMessage(MouseMessages.WM_LBUTTONDOWNCLK)

WinHook1.Monitor = HookMonitor.EntireSystem
WinHook1.HookType = HookTypes.Mouse
AddHandler WinHook1.OnMouseHook, AddressOf WinHook1_ OnMouseHook

WinHook1.Enabled = True

[C#]
using Desaware.SpyWorks;

WinHook1.Messages = new WindowsMessageList();
WinHook1.Messages.AddMessage(MouseMessages.WM_LBUTTONDOWNCLK);

WinHook1.Monitor = HookMonitor.EntireSystem;
WinHook1.HookType = HookTypes.Mouse;
WinHook1.OnMouseHook += new
MouseHookEventHandler(WinHook1_OnMouseHook);
WinHook1.Enabled = true;

```

8. Replace your WinHook control's event with the corresponding event and move your code to the new event. The new events are described in the *Event changes* section. For the sample code above, the OnMouseHook event is as follows:

```

[VB]
Private Sub WinHook1_OnMouseHook(ByVal sender As Object, ByVal e As
Desaware.SpyWorks.MouseHookEventArgs)

[C#]
private void WinHook1_OnMouseHook(object sender,
Desaware.SpyWorks.MouseHookEventArgs e)

```

9. Refer to the Events changes section for a detail list of the differences of parameter types for each individual event.
10. Remove the event handler when done using the winhook object.

```

[VB]
RemoveHandler WinHook1.OnMouseHook, AddressOf WinHook1_OnMouseHook
[C#]
WinHook1.OnMouseHook -= new
MouseHookEventHandler(WinHook1_OnMouseHook);

```

Property changes:

ClearMessage	Obsolete. Use the Messages.RemoveMessage method to
--------------	--

	remove a Windows message from the list of messages being hooked.
CurrentProcessFlag	Obsolete. Use MessageHookEventArgs.process to determine if the message was intercepted from another process or not when the HookType is set to CallWndProc or CallWndProcRet. Use MessageHookEventArgs.handling to determine if the message was actually removed from the queue, or just peeked without removal when the HookType is set to GetMessage.
HookEnabled	Replaced by the Enabled property.
HookType	<p>Changed from the Desaware.SpyWorks.dwshkNET.HookTypeConstants data type to the Desaware.SpyWorks.HookTypes data type. The Desaware.SpyWorks.HookTypes includes 4 new hook types:</p> <p>Desaware.SpyWorks.HookTypes.ForegroundIdle, Desaware.SpyWorks.HookTypes.Keyboard, Desaware.SpyWorks.HookTypes.KeyboardLL, and Desaware.SpyWorks.HookTypes.MouseLL. (The keyboard hook types can only be set in the KeyHook component that derives from the WinHook component).</p>
HwndParam	Changed from an Integer data type to an IntPtr data type. You can use the IntPtr.op_Explicit function to cast an Integer data type to an IntPtr data type, or create a new IntPtr object with an integer as the constructor parameter.
MessageArray	Obsolete. Use the Messages.Messages property to retrieve an array of the messages being hooked.
MessageCount	Obsolete. Use the Messages.Messages.Length property to retrieve the number of messages being hooked.
Messages	The data type for this property has changed. It use to be such that you were able to assign a messages by setting the Messages in design time or at run time by setting the Messages property to a message number. Now Messages refer to a WindowsMessageList data type (an arraylist). This property is initially set to Nothing which will hook all Windows messages. Most Windows messages are also exposed through one of the Desaware.SpyWorks.*Messages enumerator.
Monitor	Changed from the Desaware.SpyWorks.dwshkNET.MonitorConstants data type to

	the Desaware.SpyWorks.WinHook.Monitor data type. The Desaware.SpyWorks.WinHook.Monitor enumerator does not support the MySiblings and ThisForm constants anymore (since the component is no longer associated with a particular form).
Notify	Replaced with AsyncNotification. Refer to the Fundamental Differences section early in this document for information on regarding threading changes for the Windows Hook object.
PostEvent	Obsolete. Use an asynchronous delegate if you need to defer an operation.
PostOnFreeze	Obsolete. Events are never “frozen”
PostOnFreezeMax	Obsolete.
RegMessage	Obsolete. Use the GetRegisteredWindowMessage property to retrieve a unique Windows message number on the current system for the specified string. Then use the Messages.AddMessage to include the message to hook.
RegMessageNum	Obsolete. Use the GetRegisteredWindowMessage property to retrieve a unique Windows message number on the current system for the specified string.
TaskParam	Replaced by ProcessParam.
UseDirectInterface	Obsolete.

Event changes:

CBTProc	Replaced by the OnCBTHook event. This event’s AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_CBTProcEvent type is replaced by the CBTHookEventArgs type that exposes similar fields. The <i>lp</i> and <i>wp</i> fields are now <i>lParam</i> and <i>wParam</i> respectively. The <i>code</i> field is now a CBTMessageType enumerator type.
DelayedEvent	Obsolete.
JournalPlayProc	Replaced by the OnJournalPlaybackHook event. This event’s AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_JournalPlayProcEvent type is replaced by the JournalPlaybackHookEventArgs type that exposes the same fields. The exception is the <i>wnd</i> field that is now an IntPtr type instead of

	an Integer type and the <i>code</i> field is now a <i>JournalMessageType</i> enumerator type.
JournalRecordProc	Replaced by the OnJournalRecordHook event. This event's <i>AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_JournalRecordProcEvent</i> type is replaced by the <i>JournalRecordHookEventArgs</i> type that exposes the same fields. The exception is the <i>wnd</i> field that is now an <i>IntPtr</i> type instead of an Integer type and the <i>code</i> field is now a <i>JournalMessageType</i> enumerator type.
MessageProc	Replaced by the OnMessageHook event. This event's <i>AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MessageProcEvent</i> type is replaced by the <i>MessageHookEventArgs</i> type that exposes similar fields. The <i>wnd</i> field has been replaced by the <i>hwnd</i> field and is now an <i>IntPtr</i> type instead of an Integer type. The <i>src</i> field has been replaced by the <i>source</i> field and is now a <i>MessageInputSource</i> enumerator type.
MouseProc	Replaced by the OnMouseHook event. This event's <i>AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MouseProcEvent</i> type is replaced by the <i>MouseHookEventArgs</i> type that exposes similar fields. The <i>wnd</i> field has been replaced by the <i>hwnd</i> field and is now an <i>IntPtr</i> type instead of an Integer type. The <i>peek</i> field has been replaced by the <i>handling</i> field and is now a <i>MessageHandling</i> enumerator type.
ShellProc	Replaced by the OnShellHook event. This event's <i>AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_ShellProcEvent</i> type is replaced by the <i>ShellHookEventArgs</i> type that exposes similar fields. The <i>lp</i> and <i>wp</i> fields are now <i>lParam</i> and <i>wParam</i> respectively. The <i>code</i> field is now a <i>ShellMessageType</i> enumerator type.
WndMessage	Replaced by the OnMessageHook event. This event's <i>AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_WndMessageEvent</i> type is replaced by the <i>MessageHookEventArgs</i> type that exposes similar fields. The <i>wnd</i> field has been replaced by the <i>hwnd</i> field and is now an <i>IntPtr</i> type instead of an Integer type.
WndMessageRet	Replaced by the OnMessageHook event. This event's <i>AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_WndMessageRetEvent</i> type is replaced by the <i>MessageHookEventArgs</i> type that exposes similar fields. The <i>wnd</i> field has been replaced by the <i>hwnd</i> field and is now an <i>IntPtr</i> type instead of an Integer type.

Migrating the WinHook control from a .NET project – KeyBoard hook migration

1. Write down the design time properties for each of the winhook controls, especially keys that the control is detecting.
2. Delete all winhook controls from all forms - you may also need to remove a line of code from the form the control is on similar to:

```
[VB]
Friend WithEvents WinHook1 As AxDesaware.SpyWorks.dwshkNET.AxWinHook
```

```
[C#]
private AxDesaware.SpyWorks.dwshkNET.AxWinHook WinHook1;
```

The library may be AxDWSHK80Lib if you are migrating from the SpyWorks 8 ActiveX controls.

3. Remove the project reference to Desaware.SpyWorks.dwshkNet or DWSHK80LIB and AxDWSHK80LIB (or DWSBC36LIB if migrating from SpyWorks 7.1 or earlier).
4. Add a project reference to the new Desaware.shcomponent.dll .NET component.
5. Declare the new WinHook object. In VB .NET, you can use the WithEvents declaration instead of adding events explicitly.

```
[VB]
Friend KeyHook1 As Desaware.SpyWorks.KeyHook
[C#]
internal Desaware.SpyWorks.KeyHook KeyHook1;
```

6. Add code to create a new instance of the object.

```
[VB]
KeyHook1 = New Desaware.SpyWorks.KeyHook()
[C#]
KeyHook1 = new Desaware.SpyWorks.KeyHook();
```

7. Write code to restore the previous WinHook control's design time properties after creating the KeyHook object. Usually, you would just set the Keys and KeyboardHook properties for the WinHook control. These properties have been changed for the new KeyHook object. Refer to the *Property changes* section for more information.

```
[VB]
Imports Desaware.SpyWorks

KeyHook1.HookType = HookTypes.Keyboard
```

```

KeyHook1.Monitor = HookMonitor.EntireSystem
KeyHook1.KeyFilterList = New KeyList()
' Add the Del and Enter keys to be detected
KeyHook1.KeyFilterList.AddKey(VirtualKeys.VK_Delete, KeyFlags.None)
KeyHook1.KeyFilterList.AddKey(VirtualKeys.VK_Enter, KeyFlags.None)
AddHandler KeyHook1.OnKeyDown, AddressOf KeyHook1_OnKeyDown

```

```

KeyHook1.Enabled = True

```

```

[C#]

```

```

using Desaware.SpyWorks;

```

```

KeyHook1.HookType = HookTypes.Keyboard;
KeyHook1.Monitor = HookMonitor.EntireSystem;
KeyHook1.KeyFilterList = new KeyList();
// Add the Del and Enter keys to be detected
KeyHook1.KeyFilterList.AddKey(VirtualKeys.VK_Delete, KeyFlags.None)
KeyHook1.KeyFilterList.AddKey(VirtualKeys.VK_Enter, KeyFlags.None)
// add event handler
KeyHook1.OnKeyDown += new
KeyDownHookEventHandler(KeyHook1_OnKeyDown);
KeyHook1.Enabled = true;

```

8. Replace your WinHook control's KeyDownHook or KeyUpHook event with the OnKeyDown or OnKeyUp event and move your code to the new event. These events are declared as follows:

```

[VB]

```

```

Private Sub KeyHook1_OnKeyDown(ByVal sender As Object, ByVal e As
Desaware.SpyWorks.KeyboardHookEventArgs)

```

```

Private Sub KeyHook1_OnKeyUp(ByVal sender As Object, ByVal e As
Desaware.SpyWorks.KeyboardHookEventArgs)

```

```

[C#]

```

```

private void KeyHook1_OnKeyDown(object sender,
Desaware.SpyWorks.KeyboardHookEventArgs e)

```

```

private void KeyHook1_OnKeyUp(object sender,
Desaware.SpyWorks.KeyboardHookEventArgs e)

```

9. The KeyboardHookEventArgs data type found in the OnKeyDown/OnKeyUp events is similar to the AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyDownHookEvent and AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyUpHookEvent data types. The exception is the KeyboardHookEventArgs.discard field that is now a Boolean instead of a Short (16 bit integer).

10. Remove the OnKeyDown or OnKeyUp event handlers when done using the keyhook object.

```

[VB]

```

```

RemoveHandler KeyHook1.OnKeyDown, AddressOf KeyHook1_OnKeyDown
RemoveHandler KeyHook1.OnKeyUp, AddressOf KeyHook1_OnKeyUp

```

```
[C#]
KeyHook1.OnKeyDown -= new KeyDownHookEventHandler(KeyHook1_OnKeyDown);
KeyHook1.OnKeyUp -= new KeyUpHookEventHandler(KeyHook1_OnKeyUp);
```

Property changes:

(Note: This class inherits from the WinHook class, and thus supports additional methods and properties via inheritance)

ClearKey	Obsolete. Use the KeyFilterList.RemoveKey method to remove a key from the list of keys being hooked.
KeyArray	Obsolete. Use the KeyFilterList.Keys property to retrieve an array of the keys being hooked.
KeyboardEvent	Obsolete. The old KbdHook style is no longer supported, use the KeyDown and KeyUp events instead.
KeyboardHook	Obsolete. Use the Monitor property to specify the scope of the keyboard hook and the Enabled property to turn the hook on and off.
KeyboardNotify	Obsolete. Use the AsyncNotification property to specify whether the KeyHook should trigger an event when the key is detected or trigger an event asynchronously at a later time.
KeyCount	Obsolete. Use the KeyFilterList.Count property to retrieve the number of keys being hooked.
KeyIgnoreCapsLock	Obsolete.
Keys	Use the KeyFilterList.AddKey method to dynamically add a new key to be hooked.
KeyViewPeeked	Replaced by the ViewPeeked property.
TaskParam	Replaced by the ProcessParam property.

Event changes:

KbdHook	Obsolete. The old KbdHook style is no longer supported, use the OnKeyDown and OnKeyUp events instead.
---------	---

KeyDownHook	Replaced by the OnKeyDown event. This event's AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyDownHookEvent type is replaced by the KeyboardHookEventArgs type that exposes the same fields. The exception is the <i>discard</i> field that is now a Boolean type instead of a Short type. The KeyboardHookEventArgs also exposes some new functions to return information from the <i>keycode</i> field.
KeyUpHook	Replaced by the OnKeyUp event. This event's AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyUpHookEvent type is replaced by the KeyboardHookEventArgs type that exposes the same fields. The exception is the <i>discard</i> field that is now a Boolean type instead of a Short type. The KeyboardHookEventArgs also exposes some new functions to return information from the <i>keycode</i> field.

Desaware.shcomponent.dll Reference

Introduction

The Desaware.shcomponent.dll component is a native .NET component that exposes Windows Subclass and Windows Hooks technology previously found in the SpyWorks Subclass control and SpyWorks WinHook control. Desaware recommends using the new Desaware.shcomponent.dll component in place of the previous Subclass and WinHook COM controls for development on the Visual Studio .NET platform.

The following is detailed reference information for the Desaware.shcomponent.dll classes. Properties and methods of base classes are not listed unless they are overridden.

Note on thread safety:

Unless otherwise noted, as is common in the .NET framework, static methods of objects are thread safe. Instance methods are not.

Events are generally not thread safe (i.e. they can be raised on any thread). Events raised when asynchronous notification is requested are raised on a thread from the .NET thread pool, and are not synchronized to any window or other thread (this is standard .NET framework behavior). If you access form or control properties from event handling code, be sure to use the Control class InvokeRequired property to determine if you must use the InvokeMember method to access the control property.

Desaware.SpyWorks Enumerators

CBTMessageType	Used by the <i>code</i> field of the CBTHookEventArgs object. Describes the type of code that triggered an OnCBTHook event. Refer to your Windows API documentation for the CBTProc
----------------	---

	<p>function for information on these fields.</p> <p>Activate - 5 ClickSkipped - 6 CreateWnd - 3 DestroyWnd - 4 KeySkipped - 7 MixMax - 1 MoveSize - 0 QueueSync - 2 SetFocus - 9 SysCommand - 8</p>
HookMonitor	<p>Used by the WinHook object's <i>Monitor</i> property. Refer to the <i>Monitor</i> property for detailed information on these fields.</p> <p>EntireSystem - 4 HwndAndChildren - 6 HwndParam - 5 ProcessParam - 3 ThisProcess - 2 ThisThread - 0 ThreadParam - 1</p>
HookTypes	<p>Used by the WinHook and KeyHook objects <i>HookType</i> property. Refer to the <i>HookType</i> property for these objects for detailed information on these fields.</p> <p>CallWndProc - 4 CallWndProcRet - 9 CBT - 5 ForegroundIdle - 13 GetMessage - 0 JournalPlayback - 7 JournalRecord - 6 Keyboard - 10 KeyboardLL - 11 MessageFilter - 2 Mouse - 1 MouseLL - 12 Shell - 8 SysMessageFilter - 3</p>
JournalMessageType	<p>Used by the <i>code</i> field of the JournalPlaybackHookEventArgs and JournalRecordHookEventArgs objects. Describes the type of code that triggered an OnJournalPlaybackHook or</p>

	<p>OnJournalRecordHook event. Refer to your Windows API documentation for the JournalPlaybackProc or JournalRecordProc function for information on these fields.</p> <p>Action - 0 GetNext - 1 NoRemove - 3 Skip - 2 SysModalOff - 5 SysModalOn - 4</p>
KeyboardHookType	<p>Used by the <i>code</i> field of the KeyboardHookEventArgs object. Describes the type of code that triggered an OnKeyDown or OnKeyUp event. <i>Action</i> refers to a normal keystroke message event. <i>NoRemove</i> indicates that the keystroke message has not been removed from the message queue. (An application called the PeekMessage function, specifying the PM_NOREMOVE flag.) For most cases, you can just ignore this event if <i>NoRemove</i> is set.</p> <p>Action - 0 NoRemove - 3</p>
KeyFlags	<p>Used by the KeyList object to specify modifier keys. The values of these modifier keys correspond to their Windows API values.</p> <p>Alt – 262144 (hex 40000) Ctrl – 131072 (hex 20000) None - 0 Shift – 65536 (hex 10000)</p>
MessageHandling	<p>Used by the <i>handling</i> field of the MessageHookEventArgs object. Describes the type of code that triggered an OnMessageHook event. <i>Remove</i> refers to a normal windows message event. <i>NoRemove</i> indicates that the windows message has not been removed from the message queue. (An application called the PeekMessage function, specifying the PM_NOREMOVE flag.) For most cases, you can just ignore this event if <i>NoRemove</i> is set.</p> <p>NoRemove - 0 Remove - 1</p>
MessageInputSource	<p>Used by the <i>source</i> field of the MessageHookEventArgs object. Describes the type of input event that generated the windows message when the WinHook object's <i>HookType</i> property is set</p>

	<p>to MessageFilter or SysMessageFilter. The values of these fields correspond to their Windows API values.</p> <p>DDE – 32769 (hex 8001) Dialog - 0 Menu - 2 Scrollbar - 5</p>
ShellMessageType	<p>Used by the <i>code</i> field of the ShellHookEventArgs object. Describes the type of code that triggered an OnShellHook event. Refer to your Windows API documentation for the ShellProc function for information on these fields.</p> <p>AccessibilityState - 11 ActivateShellWindow - 3 AppCommand - 12 GetMinRect - 5 Language - 8 Redraw - 6 TaskMan - 7 WindowActivated - 4 WindowCreated - 1 WindowDestroyed - 2 WindowReplaced - 13</p>
SubclassingTypes	<p>Used by the <i>SubclassingType</i> property of the Subclasser object. Refer to the <i>SubclassingType</i> property for more detailed information on these fields.</p> <p>Asynchronous - 2 PostDefault - 1 PreDefault - 0</p>

Desaware.SpyWorks Minor Classes

These classes are mainly used to expose corresponding Windows API structures or constants. The Keys and Messages fields correspond to their respective Windows API values.

ButtonControlMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>BM_GETCHECK – 240 BM_GETSTATE – 242 BM_SETCHECK – 241 BM_SETSTATE – 243</p>
-----------------------	---

	BM_SETSTYLE – 244
CBTActivateStruct	<p>Contains window activation information for a CBTHook.</p> <p>fMouse – Boolean – True if the window is activated due to a mouse click. hWndActivate – IntPtr – Window handle of the active window.</p>
CBTCreateStruct	<p>Contains window creation information for a CBTHook. Refer to your Windows API documentation on CBT_CREATEWND for information on these fields.</p> <p>cx - Integer cy - Integer dwExStyle - Integer hInstance - Integer hMenu - Integer hWndAfter - IntPtr hwndParent - IntPtr lpCreateParams - Integer lpszClass - String lpszName - String style - Integer x - Integer y - Integer</p>
CBTHookEventArgs	<p>Contains the parameters relevant to the OnCBTHook event. Refer to the OnCBTHook event for an explanation of each field.</p> <p>ActivateStruct - CBTActivateStruct BlockCBTOperation - Boolean code - CBTMessageType CreateWndStruct - CBTCreateStruct lParam - Integer MouseHookStruct - CBTMouseHookStruct MoveSizeRect - Rectangle nDef - Short wParam - Integer</p>
CBTMouseHookStruct	<p>Contains mouse information for a CBTHook. Refer to your Windows API documentation on MOUSEHOOKSTRUCT for information on these</p>

	<p>fields.</p> <p>ExtraInfo - Integer HitTestCode - Integer hWnd - IntPtr pt - Point</p>
ClipboardMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>WM_ASKCBFORMATNAME - 780 WM_CHANGECHAIN - 781 WM_CLEAR - 771 WM_COPY - 769 WM_CUT - 768 WM_DESTROYCLIPBOARD - 775 WM_DRAWCLIPBOARD - 776 WM_HSCROLLCLIPBOARD - 782 WM_PAINTCLIPBOARD - 777 WM_PASTE - 770 WM_RENDERALLFORMATS - 774 WM_RENDERFORMAT - 773 WM_SIZECLIPBOARD - 779 WM_UNDO - 772 WM_VSCROLLCLIPBOARD - 778</p>
ComboboxControlMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>CB_ADDSTRING - 323 CB_DELETESTRING - 324 CB_DIR - 325 CB_FINDSTRING - 332 CB_FINDSTRINGEXACT - 344 CB_GETCOUNT - 326 CB_GETCURSEL - 327 CB_GETDROPPEDCONTROLRECT - 338 CB_GETDROPPEDSTATE - 343 CB_GETEDITSEL - 320 CB_GETEXTENDEDUI - 342 CB_GETITEMDATA - 336 CB_GETITEMHEIGHT - 340 CB_GETLBTEXT - 328 CB_GETLBTEXTLEN - 329 CB_GETLOCALE - 346 CB_INSERTSTRING - 330</p>

	<p> CB_LIMITTEXT - 321 CB_RESETCONTENT - 331 CB_SELECTSTRING - 333 CB_SETCURSEL - 334 CB_SETEDITSEL - 322 CB_SETEXTENDEDUI - 341 CB_SETITEMDATA - 337 CB_SETITEMHEIGHT - 339 CB_SETLOCALE - 345 CB_SHOWDROPDOWN - 335 </p>
CtlColorMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p> WM_CTLCOLOR - 25 WM_CTLCOLORBTN - 309 WM_CTLCOLORDLG - 310 WM_CTLCOLOREDIT - 307 WM_CTLCOLORLISTBOX - 308 WM_CTLCOLORMSGBOX - 306 WM_CTLCOLORSCROLLBAR - 311 WM_CTLCOLORSTATIC - 312 </p>
DDEMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p> WM_DDE_ACK - 996 WM_DDE_ADVISE - 994 WM_DDE_DATA - 997 WM_DDE_EXECUTE - 1000 WM_DDE_INITIATE - 992 WM_DDE_POKE - 999 WM_DDE_REQUEST - 998 WM_DDE_TERMINATE - 993 WM_DDE_UNADVISE - 995 </p>
EditControlMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p> EM_CANUNDO - 198 EM_EMPTYUNDOBUFFER - 205 EM_FMTLINES - 200 EM_GETFIRSTVISIBLELINE - 206 EM_GETHANDLE - 189 EM_GETLINE - 196 EM_GETLINECOUNT - 186 </p>

	EM_GETMODIFY - 184 EM_GETPASSWORDCHAR - 210 EM_GETRECT - 178 EM_GETSEL - 176 EM_GETTHUMB - 190 EM_GETWORDBREAKPROC - 209 EM_LIMITTEXT - 197 EM_LINEFROMCHAR - 201 EM_LINEINDEX - 187 EM_LINELENGTH - 193 EM_LINESCROLL - 182 EM_REPLACESEL - 194 EM_SCROLL - 181 EM_SCROLLCARET - 183 EM_SETHANDLE - 188 EM_SETMODIFY - 185 EM_SETPASSWORDCHAR - 204 EM_SETREADONLY - 207 EM_SETRECT - 179 EM_SETRECTNP - 180 EM_SETSEL - 177 EM_SETTABSTOPS - 203 EM_SETWORDBREAKPROC - 208 EM_UNDO - 199
FileManagerMessages	Refer to your Windows API documentation on these messages for information on these fields. FM_GETDRIVEINFO - 1537 FM_GETFILESEL - 1540 FM_GETFILESELLFN - 1541 FM_GETFOCUS - 1536 FM_GETSELCOUNT - 1538 FM_GETSELCOUNTLFN - 1539 FM_REFRESH_WINDOWS - 1542 FM_RELOAD_EXTENSIONS - 1543
ForegroundIdleHookEventArgs	Contains the parameters relevant to the OnForegroundIdleHook event. Refer to the OnForegroundIdleHook event for an explanation of each field. nodef - Short
FunctionKeys	Function_F1 (value = 112) to Function_F24 (value = 135)

JournalPlaybackHookEventArgs	<p>Contains the parameters relevant to the OnJournalPlaybackHook event. Refer to the OnJournalPlaybackHook event for an explanation of each field.</p> <p>code - JournalMessageType delay - Integer msg - Integer mtime - Integer paramH - Integer paramL - Integer wnd - IntPtr</p>
JournalRecordHookEventArgs	<p>Contains the parameters relevant to the OnJournalRecordHook event. Refer to the OnJournalRecordHook event for an explanation of each field.</p> <p>code - JournalMessageType msg - Integer mtime - Integer nodef - Short paramH - Integer paramL - Integer wnd - IntPtr</p>
KeyboardHookEventArgs	<p>Contains the parameters and helper methods relevant to the OnKeyDown or OnKeyUp events. Refer to the OnKeyDown or OnKeyUp events for an explanation of each field.</p> <p>Function GetScanCode() As Integer Function IsAltPressed() As Boolean Function IsExtended() As Boolean Function IsKeyRelease() As Boolean Function IsPreviouslyPressed() As Boolean code - KeyboardHookType discard - Boolean keycode - Integer keystate - Short processId - Integer repetitions - Short shiftstate - Short</p>
KeyboardLLHookEventArgs	Contains the parameters and helper methods relevant

	<p>to the OnKeyDownLL or OnKeyUpLL events. Refer to the OnKeyDownLL or OnKeyUpLL events for an explanation of each field.</p> <p>Function GetScanCode() As Integer Function IsAltPressed() As Boolean Function IsExtended() As Boolean Function IsInjected() As Boolean Function IsKeyRelease() As Boolean discard - Boolean ExtraInfo - Integer flags - Integer message - Integer processId - Integer scancode - Integer shiftstate - Short time - Integer vkCode - Integer</p>
KeyboardMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>WM_CHAR - 258 WM_DEADCHAR - 259 WM_KEYDOWN - 256 WM_KEYUP - 257 WM_SYSCHAR - 262 WM_SYSDEADCHAR - 263 WM_SYSKEYDOWN - 260 WM_SYSKEYUP - 261</p>
KeyDefinitions	<p>Base class from which the *Keys classes inherit from, you will not need to use this class directly.</p>
KeyList	<p>Used by the KeyHook object's KeyFilterList property to hold a list of keys to detect. If the KeyFilterList property is not set, or is set to an instance of this class that does not contain any keys, the KeyHook object will detect all keys.</p> <p>AddKey – Adds the key specified in <i>keyvalue</i> to the list of keys to detect. The <i>flags</i> parameter specifies modifier keys for the key to add.</p> <pre>[VB] Sub AddKey(ByVal keyvalue As Integer, ByVal flags As Desaware.SpyWorks.KeyFlags) [C#] void AddKey(int keyvalue, Desaware.SpyWorks.KeyFlags flags)</pre>

	<p>Contains – Returns whether this key list contains the key specified in <i>keyfilter</i>. The <i>keyfilter</i> parameter is a 32 bit value that contains the virtual key code in the lower 16 bits and the key modifier (<i>KeyFlags</i>) in the upper 16 bits.</p> <p>[VB] Function Contains(ByVal keyfilter As Integer) As Boolean [C#] bool Contains(int keyfilter)</p> <p>Count – Returns the number of keys being detected. A 0 value indicates that all keys will be detected.</p> <p>[VB] Function Count() As Integer [C#] int Count()</p> <p>Keys – Returns an integer array containing the values of the keys being detected. Each value in the array is a 32 bit value that contains the virtual key code in the lower 16 bits and the key modifier (<i>KeyFlags</i>) in the upper 16 bits.</p> <p>[VB] Function Keys() As Integer() [C#] int() Keys()</p> <p>RemoveKey – Removes the key specified in <i>keyvalue</i> from the list of keys to detect. The <i>flags</i> parameter specifies modifier keys for the key to remove.</p> <p>[VB] Sub RemoveKey(ByVal keyvalue As Integer, ByVal flags As Desaware.SpyWorks.KeyFlags) [C#] void RemoveKey(int keyvalue, Desaware.SpyWorks.KeyFlags flags)</p>
LetterKeys	LTR_0 (value = 48) to LTR_9 (value = 57) and LTR_A (value = 65) to LTR_Z (value = 90)
ListboxControlMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>LB_ADDFILE - 406 LB_ADDSTRING - 384 LB_DELETESTRING - 386 LB_DIR - 397 LB_FINDSTRING - 399 LB_FINDSTRINGEXACT - 418 LB_GETANCHORINDEX - 413 LB_GETCARETINDEX - 415 LB_GETCOUNT - 395 LB_GETCURSEL - 392</p>

	<p> LB_GETHORIZONTALEXTENT - 403 LB_GETITEMDATA - 409 LB_GETITEMHEIGHT - 417 LB_GETITEMRECT - 408 LB_GETLOCALE - 422 LB_GETSEL - 391 LB_GETSELCOUNT - 400 LB_GETSELITEMS - 401 LB_GETTEXT - 393 LB_GETTEXTLEN - 394 LB_GETTOPINDEX - 398 LB_INSERTSTRING - 385 LB_RESETCONTENT - 388 LB_SELECTSTRING - 396 LB_SELITEMRANGE - 411 LB_SELITEMRANGEEX - 387 LB_SETANCHORINDEX - 412 LB_SETCARETINDEX - 414 LB_SETCOLUMNWIDTH - 405 LB_SETCOUNT - 433 LB_SETCURSEL - 390 LB_SETHORIZONTALEXTENT - 404 LB_SETITEMDATA - 410 LB_SETITEMHEIGHT - 416 LB_SETLOCALE - 421 LB_SETSEL - 389 LB_SETTABSTOPS - 402 LB_SETTOPINDEX - 407 </p>
ListviewControlMessages	<p> Refer to your Windows API documentation on these messages for information on these fields. </p> <p> LVM_APPROXIMATEVIEWRECT - 4160 LVM_ARRANGE - 4118 LVM_CREATEDRAGIMAGE - 4129 LVM_DELETEALLITEMS - 4105 LVM_DELETECOLUMN - 4124 LVM_DELETEITEM - 4104 LVM_EDITLABEL - 4119 LVM_ENSUREVISIBLE - 4115 LVM_FINDITEM - 4109 LVM_FIRST - 4096 LVM_GETBKCOLOR - 4096 LVM_GETBKIMAGE - 4165 LVM_GETCALLBACKMASK - 4106 LVM_GETCOLUMN - 4121 </p>

LVM_GETCOLUMNORDERARRAY - 4155
LVM_GETCOLUMNWIDTH - 4125
LVM_GETCOUNTPERPAGE - 4136
LVM_GETEDITCONTROL - 4120
LVM_GETEXTENDEDLISTVIEWSTYLE - 4151
LVM_GETHEADER - 4127
LVM_GETHOTCURSOR - 4159
LVM_GETHOTITEM - 4157
LVM_GETHOVERTIME - 4168
LVM_GETIMAGELIST - 4098
LVM_GETISEARCHSTRING - 4148
LVM_GETITEM - 4101
LVM_GETITEMCOUNT - 4100
LVM_GETITEMPOSITION - 4112
LVM_GETITEMRECT - 4110
LVM_GETITEMSPACING - 4147
LVM_GETITEMSTATE - 4140
LVM_GETITEMTEXT - 4141
LVM_GETNEXTITEM - 4108
LVM_GETNUMBEROFWORKAREAS - 4169
LVM_GETORIGIN - 4137
LVM_GETSELECTEDCOUNT - 4146
LVM_GETSELECTIONMARK - 4162
LVM_GETSTRINGWIDTH - 4113
LVM_GETSUBITEMRECT - 4152
LVM_GETTEXTBKCOLOR - 4133
LVM_GETTEXTCOLOR - 4131
LVM_GETTOOLTIPS - 4174
LVM_GETTOPINDEX - 4135
LVM_GETUNICODEFORMAT - 8198
LVM_GETVIEWRECT - 4130
LVM_GETWORKAREAS - 4166
LVM_HITTEST - 4114
LVM_INSERTCOLUMN - 4123
LVM_INSERTITEM - 4103
LVM_REDRAWITEMS - 4117
LVM_SCROLL - 4116
LVM_SETBKCOLOR - 4097
LVM_SETBKIMAGE - 4164
LVM_SETCALLBACKMASK - 4107
LVM_SETCOLUMN - 4122
LVM_SETCOLUMNORDERARRAY - 4154
LVM_SETCOLUMNWIDTH - 4126
LVM_SETEXTENDEDLISTVIEWSTYLE - 4150
LVM_SETHOTCURSOR - 4158
LVM_SETHOTITEM - 4156

	LVM_SETHOVERTIME - 4167 LVM_SETICONSPACING - 4149 LVM_SETIMAGELIST - 4099 LVM_SETITEM - 4102 LVM_SETITEMCOUNT - 4143 LVM_SETITEMPOSITION - 4111 LVM_SETITEMPOSITION32 - 4145 LVM_SETITEMSTATE - 4139 LVM_SETITEMTEXT - 4142 LVM_SETSELECTIONMARK - 4163 LVM_SETTEXTBKCOLOR - 4134 LVM_SETTEXTCOLOR - 4132 LVM_SETTOOLTIPS - 4170 LVM_SETUNICODEFORMAT - 8197 LVM_SETWORKAREAS - 4161 LVM_SORTITEMS - 4144 LVM_SUBITEMHITTEST - 4153 LVM_UPDATE - 4138
MdiMessages	Refer to your Windows API documentation on these messages for information on these fields. WM_DROPFILES - 563 WM_MDIACTIVATE - 546 WM_MDICASCADE - 551 WM_MDICREATE - 544 WM_MDIDESTROY - 545 WM_MDIGETACTIVE - 553 WM_MDIICONARRANGE - 552 WM_MDIMAXIMIZE - 549 WM_MDINEXT - 548 WM_MDIREFRESHMENU - 564 WM_MDIRESTORE - 547 WM_MDISETMENU - 560 WM_MDITILE - 550
MessageHookEventArgs	Contains the parameters relevant to the OnMessageHook event. Refer to the OnMessageHook event for an explanation of each field. handling - MessageHandling hwnd - IntPtr inproccall - Boolean lp - Integer msg - Integer nodef - Short

	<p>process - Integer retval - Integer source - MessageInputSource wp - Integer</p>
MiscellaneousMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>WM_CHOOSEFONT_GETLOGFONT - 1025 WM_CPL_LAUNCH - 5120 WM_CPL_LAUNCHED - 5121 WM_IME_CHAR - 646 WM_IME_COMPOSITION - 271 WM_IME_COMPOSITIONFULL - 644 WM_IME_CONTROL - 643 WM_IME_ENDCOMPOSITION - 270 WM_IME_KEYDOWN - 656 WM_IME_KEYLAST - 271 WM_IME_KEYUP - 657 WM_IME_NOTIFY - 642 WM_IME_REQUEST - 648 WM_IME_SELECT - 645 WM_IME_SETCONTEXT - 641 WM_IME_STARTCOMPOSITION - 269 WM_PSD_ENVSTAMPRECT - 1029 WM_PSD_FULLPAGERECT - 1025 WM_PSD_GREEKTEXTRECT - 1028 WM_PSD_MARGINRECT - 1027 WM_PSD_MINMARGINRECT - 1026 WM_PSD_PAGESETUPDLG - 1024 WM_PSD_YAFULLPAGERECT - 1030</p>
MouseHookEventArgs	<p>Contains the parameters relevant to the OnMouseHook event. Refer to the OnMouseHook event for an explanation of each field.</p> <p>flags - Integer handling - MessageHandling hitcode - Integer hwnd - IntPtr mousedata - Integer msg - Integer nodef - Short process - Integer time - Integer x - Integer</p>

	xtra - Integer y - Integer
MouseMessages	Refer to your Windows API documentation on these messages for information on these fields. WM_LBUTTONDOWNBLCLK - 515 WM_LBUTTONDOWN - 513 WM_LBUTTONUP - 514 WM_MBUTTONDOWNBLCLK - 521 WM_MBUTTONDOWN - 519 WM_MBUTTONUP - 520 WM_MOUSEHOVER - 673 WM_MOUSELEAVE - 675 WM_MOUSEMOVE - 512 WM_MOUSEWHEEL - 522 WM_RBUTTONDOWNBLCLK - 518 WM_RBUTTONDOWN - 516 WM_RBUTTONUP - 517 WM_SETCURSOR - 32
MultimediaMessages	Refer to your Windows API documentation on these messages for information on these fields. MM_JOY1BUTTONDOWN - 949 MM_JOY1BUTTONUP - 951 MM_JOY1MOVE - 928 MM_JOY1ZMOVE - 930 MM_JOY2BUTTONDOWN - 950 MM_JOY2BUTTONUP - 952 MM_JOY2MOVE - 929 MM_JOY2ZMOVE - 931 MM_MCMINOTIFY - 953 MM_MIM_CLOSE - 962 MM_MIM_DATA - 963 MM_MIM_ERROR - 965 MM_MIM_LONGDATA - 964 MM_MIM_LONGERROR - 966 MM_MIM_OPEN - 961 MM_MOM_CLOSE - 968 MM_MOM_DONE - 969 MM_MOM_OPEN - 967 MM_WIM_CLOSE - 959 MM_WIM_DATA - 960 MM_WIM_OPEN - 958 MM_WOM_CLOSE - 956

	MM_WOM_DONE - 957 MM_WOM_OPEN - 955
NonClientMessages	Refer to your Windows API documentation on these messages for information on these fields. WM_GETDLGCODE - 135 WM_NCACTIVATE - 134 WM_NCCALCSIZE - 131 WM_NCCREATE - 129 WM_NCDESTROY - 130 WM_NCHITTEST - 132 WM_NCLBUTTONDBLCLK - 163 WM_NCLBUTTONDOWN - 161 WM_NCLBUTTONUP - 162 WM_NCMBUTTONDBLCLK - 169 WM_NCMBUTTONDOWN - 167 WM_NCMBUTTONUP - 168 WM_NCMOUSEMOVE - 160 WM_NCPAINT - 133 WM_NCRBUTTONDBLCLK - 166 WM_NCRBUTTONDOWN - 164 WM_NCRBUTTONUP - 165
NumericPadKeys	NumericPad_0 (value = 96) to NumericPad_9 value = 105) NumericPad_Add - 107 NumericPad_Comma - 108 NumericPad_Divide - 111 NumericPad_Multiply - 106 NumericPad_Period - 110 NumericPad_Subtract - 109
PaletteMessages	Refer to your Windows API documentation on these messages for information on these fields. WM_PALETTECHANGED - 785 WM_PALETTEISCHANGING - 784 WM_QUERYNEWPALETTE - 783
PenWindowMessages	Refer to your Windows API documentation on these messages for information on these fields. WM_GLOBALRCCHANGE - 899 WM_HEDITCTL - 901 WM_HOOKRCRESULT - 898

	<p>WM_PENWINFIRST - 896 WM_PENWINLAST - 911 WM_RCRESULT - 897 WM_SKB - 900</p>
PunctuationKeys	<p>P_Backslash - 220 P_Comma - 188 P_Dash - 189 P_Equal - 187 P_Hyphen - 192 P_LeftBracket - 219 P_Period - 190 P_Quote - 222 P_RightBracket - 221 P_Semicolon - 186 P_Slash - 191</p>
ScrollbarControlMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>SBM_ENABLE_ARROWS - 228 SBM_GETPOS - 225 SBM_GETRANGE - 227 SBM_SETPOS - 224 SBM_SETRANGE - 226 SBM_SETRANGEREDRAW - 230</p>
ShellHookEventArgs	<p>Contains the parameters relevant to the OnShellHook event. Refer to the OnShellHook event for an explanation of each field.</p> <p>MinRect - Rectangle code - ShellMessageType commandhandled - Boolean IParam - Integer nodef - Short wParam - Integer</p>
StandardMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>WM_ACTIVATE - 6 WM_ACTIVATEAPP - 28 WM_CANCELJOURNAL - 75 WM_CANCELMODE - 31 WM_CAPTURECHANGED - 533</p>

WM_CHARTOITEM - 47
WM_CHILDACTIVATE - 34
WM_CLOSE - 16
WM_COMMAND - 273
WM_COMMNOTIFY - 68
WM_COMPACTING - 65
WM_COMPAREITEM - 57
WM_CONTEXTMENU - 123
WM_COPYDATA - 74
WM_CREATE - 1
WM_DELETEITEM - 45
WM_DESTROY - 2
WM_DEVICECHANGE - 537
WM_DEVMODECHANGE - 27
WM_DISPLAYCHANGE - 126
WM_DRAWITEM - 43
WM_DROPFILES - 563
WM_ENABLE - 10
WM_ENDSESSION - 22
WM_ENTERIDLE - 289
WM_ENTERMENULOOP - 529
WM_ERASEBKGND - 20
WM_EXITMENULOOP - 530
WM_FONTCHANGE - 29
WM_GETFONT - 49
WM_GETHOTKEY - 51
WM_GETICON - 127
WM_GETMINMAXINFO - 36
WM_GETTEXT - 13
WM_GETTEXTLENGTH - 14
WM_HANDHELDFIRST - 856
WM_HANDHELDDLAST - 863
WM_HELP - 83
WM_HOTKEY - 786
WM_HSCROLL - 276
WM_ICONERASEBKGND - 39
WM_INITDIALOG - 272
WM_INITMENU - 278
WM_INITMENUPOPUP - 279
WM_INPUTLANGCHANGE - 81
WM_INPUTLANGCHANGEREQUEST - 80
WM_KILLFOCUS - 8
WM_MEASUREITEM - 44
WM_MENUCHAR - 288
WM_MENUSELECT - 287
WM_MOUSEACTIVATE - 33

WM_MOVE - 3
WM_MOVING - 534
WM_NEXTDLGCTL - 40
WM_NEXTMENU - 531
WM_NOTIFY - 78
WM_NOTIFYFORMAT - 85
WM_NULL - 0
WM_OTHERWINDOWCREATED - 66
WM_OTHERWINDOWDESTROYED - 67
WM_PAINT - 15
WM_PAINTICON - 38
WM_PARENTNOTIFY - 528
WM_POWER - 72
WM_POWERBROADCAST - 536
WM_PRINT - 791
WM_PRINTCLIENT - 792
WM_QUERYDRAGICON - 55
WM_QUERYENDSESSION - 17
WM_QUERYOPEN - 19
WM_QUEUESYNC - 35
WM_QUIT - 18
WM_SETCURSOR - 32
WM_SETFOCUS - 7
WM_SETFONT - 48
WM_SETHOTKEY - 50
WM_SETICON - 128
WM_SETREDRAW - 11
WM_SETTEXT - 12
WM_SETTINGCHANGE - 26
WM_SHOWWINDOW - 24
WM_SIZE - 5
WM_SIZING - 532
WM_SPOOLERSTATUS - 42
WM_STYLECHANGED - 125
WM_STYLECHANGING - 124
WM_SYSCOLORCHANGE - 21
WM_SYSCOMMAND - 274
WM_SYSTEMERROR - 23
WM_TCARD - 82
WM_TIMECHANGE - 30
WM_TIMER - 275
WM_USERCHANGED - 84
WM_VKEYTOITEM - 46
WM_VSCROLL - 277
WM_WINDOWPOSCHANGED - 71
WM_WINDOWPOSCHANGING - 70

	WM_WININICHANGE - 26
StaticControlMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>STM_GETICON - 369 STM_MSGMAX - 370 STM_SETICON - 368</p>
TreeviewControlMessages	<p>Refer to your Windows API documentation on these messages for information on these fields.</p> <p>TVM_CREATEDRAGIMAGE - 4370 TVM_DELETEITEM - 4353 TVM_EDITLABEL - 4366 TVM_ENDEDITLABELNOW - 4374 TVM_ENSUREVISIBLE - 4372 TVM_EXPAND - 4354 TVM_GETBKCOLOR - 4383 TVM_GETCOUNT - 4357 TVM_GETEDITCONTROL - 4367 TVM_GETIMAGELIST - 4360 TVM_GETINDENT - 4358 TVM_GETINSERTMARKCOLOR - 4390 TVM_GETSEARCHSTRING - 4375 TVM_GETITEM - 4364 TVM_GETITEMHEIGHT - 4380 TVM_GETITEMRECT - 4356 TVM_GETNEXTITEM - 4362 TVM_GETSCROLLTIME - 4386 TVM_GETTEXTCOLOR - 4384 TVM_GETTOOLTIPS - 4377 TVM_GETVISIBLECOUNT - 4368 TVM_HITTEST - 4369 TVM_INSERTITEM - 4352 TVM_SELECTITEM - 4363 TVM_SETBKCOLOR - 4381 TVM_SETIMAGELIST - 4361 TVM_SETINDENT - 4359 TVM_SETINSERTMARK - 4378 TVM_SETINSERTMARKCOLOR - 4389 TVM_SETITEM - 4365 TVM_SETITEMHEIGHT - 4379 TVM_SETSCROLLTIME - 4385 TVM_SETTEXTCOLOR - 4382 TVM_SETTOOLTIPS - 4376</p>

	<p>TVM_SORTCHILDREN - 4371 TVM_SORTCHILDRENCB - 4373</p>
VirtualKeys	<p>VK_Alt - 18 VK_Attn - 246 VK_Backspace - 8 VK_CapsLock - 20 VK_Clear - 12 VK_Ctrl - 17 VK_CtrlBreak - 3 VK_Delete - 46 VK_DownArrow - 40 VK_End - 35 VK_Enter - 13 VK_Escape - 27 VK_Execute - 43 VK_Help - 47 VK_Home - 136 VK_Insert - 45 VK_LeftArrow - 37 VK_Numlock - 144 VK_PageDown - 34 VK_PageUp - 33 VK_Pause - 19 VK_Play - 250 VK_Print - 42 VK_PrintScreen - 44 VK_Process - 229 VK_RightArrow - 39 VK_Scroll - 145 VK_Select - 41 VK_Shift - 16 VK_Space - 32 VK_Tab - 9 VK_UpArrow - 38 VK_Zoom - 251</p>
WindowsMessageList	<p>Used by the WinHook and Subclasser object's Messages property to hold a list of windows messages to detect for. If Messages property is not set, or is set to an instance of this class that does not contain any messages, then the objects will detect all windows messages.</p> <p>AddMessage – Adds the message specified to the list of messages to detect.</p>

	<pre>[VB] Sub AddMessage(ByVal messagenumber As Integer) [C#] void AddMessage(int messagenumber)</pre> <p>MessageList – Returns an integer array containing the windows messages being detected.</p> <pre>[VB] Function MessageList() As Integer() [C#] int()MessageList()</pre> <p>RemoveMessage – Removes the specified windows message from the list of windows message to detect.</p> <pre>[VB] Sub RemoveMessage (ByVal messagenumber As Integer) [C#] void RemoveMessage (int messagenumber)</pre>
WindowsMessages	Base class from which the *Messages classes inherit from, you will not need to use this class directly.
WndMessageEventArgs	<p>Contains the parameters relevant to the Subclass object's OnWndMessage event. Refer to the Subclass object's OnWndMessage event for an explanation of each field.</p> <p>hwnd - IntPtr lp - Integer msg - Integer noref - Short process - Integer retval - Integer wp - Integer</p>

Desaware.SpyWorks Main Classes

Controller Class

This class serves as the base class for the WinHook and Subclasser class. You will probably not use this class directly.

Properties

CrossTaskTimeout	<pre>[VB] Property CrossTaskTimeout As Integer [C#] int CrossTaskTimeout</pre> <p>This property contains the timeout value when</p>
------------------	---

	<p>performing cross-task (or cross-process) subclassing or system wide hooking. When doing cross-task subclassing or system wide hooking, it is quite possible for an application to freeze up the entire system. The Controller class has a watchdog timer during subclassing or hooking that, when it times out, will interrupt the subclass or hook. This property sets the timeout value. Be aware that processing time within the OnWndMessage subclass event and equivalent hook event counts toward the timeout time, so if there are any breakpoints set in these events, set this property to a very high value while debugging.</p>
--	---

Methods

<p>GetAnsiString</p>	<p>[VB] Function GetAnsiString(ByVal Address As IntPtr, ByVal Process As Integer) As String [C#] string GetAnsiString(IntPtr Address, int Process)</p> <p>This method takes an Ansi string address and the process ID, and returns the actual string. This allows the ability to retrieve a string from other processes if you know the string's address in the other process.</p>
<p>GetRegisteredWindowMessage</p>	<p>[VB] Function GetRegisteredWindowMessage (ByVal MessageName As String) As Integer [C#] int GetRegisteredWindowMessage (string MessageName)</p> <p>This method returns the message number associated with the registered Windows message specified in the MessageName parameter.</p>
<p>GetUnicodeString</p>	<p>[VB] Function GetUnicodeString(ByVal Address As IntPtr, ByVal Process As Integer) As String [C#] string GetUnicodeString (IntPtr Address, int Process)</p> <p>This method takes a Unicode string address and the process ID, and returns the actual string. This allows the ability to retrieve a string from other processes if you know the string's address in the other process.</p>

KeyHook Class

This class inherits from the WinHook class and is used to detect keyboard keys. Properties and Methods described in the WinHook class will not be described here.

Properties

HookType	<p>[VB] Property HookType As HookTypes [C#] HookTypes HookType</p> <p>Windows provides a number of different types of windows hooks. The KeyHook object supports the Keyboard and KeyboardLL hooks.</p> <p>An in-depth discussion of these hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this documentation has access to the Windows software development kit or Developer's Network CD-ROM.</p> <p>Keyboard – Implements a WH_KEYBOARD hook. This hook is triggered by keyboard events.</p> <p>KeyboardLL – Implements a WH_KEYBOARD_LL hook. This hook is triggered by keyboard events.</p> <p>This property overrides the WinHook HookType property. Attempting to set other types of hook (other than Keyboard and KeyboardLL) will cause an error when using the KeyHook class.</p>
IgnoreCapsLock	<p>[VB] Property IgnoreCapsLock As Boolean [C#] bool IgnoreCapsLock</p> <p>When set to False, alphabetic characters are shifted according to the state of the CapsLock key. Other keys are not affected. When this property has a value of True, the CapsLock key has no effect.</p>
KeyFilterList	<p>[VB] Property KeyFilterList As KeyList [C#] KeyList KeyFilterList</p> <p>This property refers to a <i>KeyList</i> object that contains the list of keys to detect. This property is initially set to Nothing which will cause the component to detect all keys. Most keys are exposed through one of the Desaware.SpyWorks.*Keys enumerators, making it easy to add them to a KeyList.</p>
ViewPeeked	<p>[VB] Property ViewPeeked As Boolean</p>

	<p>[C#] bool ViewPeeked</p> <p>Keyboard hook events are triggered any time the system reads a key event from the system queue. However, reading an event does not mean that the event is always removed from the system queue. In some cases a key event is “peeked” - previewed, and it remains in the queue. This property determines whether you want to see these peeked keys. Normally, you will want to leave this property False.</p>
--	--

Events

<p>OnKeyDown</p>	<p>[VB] Sub KeyHook1_OnKeyDown(ByVal sender As Object, ByVal e As Desaware.SpyWorks.KeyboardHookEventArgs) [C#] void KeyHook1_OnKeyDown(object sender, Desaware.SpyWorks.KeyboardHookEventArgs e)</p> <p>This event occurs when the HookType is set to Keyboard and a keyboard event occurs in which the key is pressed. The KeyboardHookEventArgs parameters are as follows:</p> <p>code (KeyboardHookType) – Indicates whether the key that triggered this event was “peeked” and will remain in the queue after this event exits. For more information, refer to the ViewPeeked property.</p> <p>discard (Boolean) – If the AsyncNotification property is False, then setting this field to True will cause the keystroke to be discarded before it is processed by Windows.</p> <p>keycode (Integer) - Specifies the virtual-key code of the key that generated the keystroke message.</p> <p>keystate (Short) – this field is defined as follows:</p> <p>Bit 0 - 7 the hardware dependent scan code. Bit 8 = 1 if this is an extended key (function key or on the numeric keypad) Bit 9 = 1 if this key is being “peeked” (i.e. this event was not removed from the system queue). This is only possible if the ViewPeeked property is set to True. Bit 13 = 1 if the ALT key is down Bit 14 = 1 if the key was previously down, 0 if it was up. Bit 15 = 1 if the key is being released, 0 if pressed.</p>
------------------	---

	<p>processId (Integer) – this field is the process id of the process receiving the keys or zero to indicate the current process.</p> <p>repetitions (Short) - this field is the repeat count for this key event.</p> <p>shiftstate (Short) – this bit field corresponds to the modifier key as follows:</p> <p>Bit 0 = 1 The shift key is down. Bit 1 = 1 The control key is down. Bit 2 = 1 The alt key is down.</p>
<p>OnKeyDownLL</p>	<pre>[VB] Sub KeyHook1_OnKeyDownLL(ByVal sender As Object, ByVal e As Desaware.SpyWorks.KeyboardLLHookEventArgs) [C#] void KeyHook1_OnKeyDownLL(object sender, Desaware.SpyWorks.KeyboardLLHookEventArgs e)</pre> <p>This event occurs when the HookType is set to KeyboardLL and a keyboard event occurs in which the key is pressed. The KeyboardLLHookEventArgs parameters are as follows:</p> <p>discard (Boolean) – If the AsyncNotification property is False, then setting this field to True will cause the keystroke to be discarded before it is processed by Windows.</p> <p>ExtraInfo (Integer) – Specifies extra information associated with this key.</p> <p>flags (Integer) – this field is defined as follows:</p> <p>Bit 0 = 1 if the key is an extended key, such as a function key or a key on the numeric keypad. The IsExtended function can be used to test this bit. Bit 4 = 1 if the key was injected. The IsInjected function can be used to test this bit. Bit 5 = 1 if the ALT key is pressed. The IsAltPressed function can be used to test this bit. NOTE that from our testing, this bit does not seem to reflect the state of the ALT key accurately some of the time. We suggest that you use the shiftstate field instead to retrieve the state of the ALT key. Bit 7 = 1 if the key is being released, 0 if pressed. The IsKeyRelease function can be used to test this bit.</p> <p>message (Integer) – Specifies the identifier of the keyboard message - one of the following KeyboardMessages fields:</p>

	<p>WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, or WM_SYSKEYUP.</p> <p>processId (Integer) – this field is the process id of the process receiving the keys or zero to indicate the current process.</p> <p>scancode (Integer) – Specifies a hardware scan code for the key that generated the keystroke message.</p> <p>shiftstate (Short) – this bit field corresponds to the modifier key as follows:</p> <p>Bit 0 = 1 The shift key is down. Bit 1 = 1 The control key is down. Bit 2 = 1 The alt key is down.</p> <p>time (Integer) – Specifies the time stamp for this message.</p> <p>vkCode (Integer) – Specifies the virtual-key code of the key that generated the keystroke message.</p>
<p>OnKeyUp</p>	<pre>[VB] Sub KeyHook1_OnKeyUp(ByVal sender As Object, ByVal e As Desaware.SpyWorks.KeyboardHookEventArgs) [C#] void KeyHook1_OnKeyUp(object sender, Desaware.SpyWorks.KeyboardHookEventArgs e)</pre> <p>This event occurs when the HookType is set to Keyboard and a keyboard event occurs in which the key is released. The parameters are identical to the OnKeyDown event.</p>
<p>OnKeyUpLL</p>	<pre>[VB] Sub KeyHook1_OnKeyUpLL(ByVal sender As Object, ByVal e As Desaware.SpyWorks.KeyboardLLHookEventArgs) [C#] void KeyHook1_OnKeyUpLL(object sender, Desaware.SpyWorks.KeyboardLLHookEventArgs e)</pre> <p>This event occurs when the HookType is set to KeyboardLL and a keyboard event occurs in which the key is released. The parameters are identical to the OnKeyDownLL event.</p>

Subclasser Class

This class allows you to subclass any existing Window. This class is more efficient than the WinHook or KeyHook classes, so you should use this class in place of those classes whenever possible.

Properties

HwndParam	<p>[VB] Property HwndParam As IntPtr [C#] IntPtr HwndParam</p> <p>Specifies the handle of the window to subclass. Set this property to zero (IntPtr.Zero) to end subclassing.</p>
Messages	<p>[VB] Property Messages As WindowsMessageList [C#] WindowsMessageList Messages</p> <p>Contains a list of the message numbers that are currently being subclassed by the Subclasser object.</p>
SubclassingType	<p>[VB] Property SubclassingType As SubclassingTypes [C#] SubclassingTypes SubclassingType</p> <p>There are three types of subclassing possible:</p> <p>SubclassingType.PreDefault – The message is intercepted before default processing for the message takes place. Any modifications you make to the message number or wParam and lParam parameters for the message will be passed on to the default message handler. You also have the option to prevent default message processing and specify a return value to the calling function.</p> <p>SubclassingType.PostDefault – The message is intercepted after default processing for the message takes place. Changes to the message number or wParam and lParam parameters will have no effect on the default message processing, but would affect any other Subclasser objects that are subclassing this window.</p> <p>SubclassingType.Asynchronous – Any time the message is detected, the OnWndMessage will be raised via an asynchronous delegate some time after the message has been processed. This method is ideal for detecting messages that do not have to be handled immediately. There are no restrictions on the contents of the event procedure when this technique is used. Changes to the message number or wParam and lParam parameters will have no effect.</p>

Events

OnWndMessage	[VB] Sub SubClass1_OnWndMessage(ByVal sender As
--------------	---

```
Object, ByVal e As  
Desaware.SpyWorks.WndMessageEventArgs)  
[C#] void SubClass1_OnWndMessage(object sender,  
Desaware.SpyWorks.WndMessageEventArgs e)
```

This event is triggered when a Windows message as specified in the Messages property is detected for the specified Window. The WndMessageEventArgs parameters are as follows:

hwnd (IntPtr) – Window handle of the window receiving the message.

lp (Integer) – The lParam parameter, this depends on the msg parameter.

msg (Integer) – The windows message number.

ndef (Short) – Set to non-zero to prevent default message processing.

process (Integer) –The process ID of the process this windows message is intended for.

retval (Integer) – The return value to the sender of the message.

wp (Integer) – The wParam parameter, this depends on the msg parameter.

When the SubclassingType property is set to PreDefault, the WndMessageEventArgs(retval) parameter only returns a value to the calling function if you set the WndMessageEventArgs.ndef parameter to non-zero. If you leave the WndMessageEventArgs.ndef parameter as zero, the default window function for the window is called and the value that it returns is passed on to the calling function.

When the SubclassingType property is set to PostDefault, the WndMessageEventArgs(retval) parameter contains the value returned by the default window function for the window. You can override this return value by changing the value of the WndMessageEventArgs(retval) parameter and setting the WndMessageEventArgs.ndef parameter to non-zero. Remember, the WndMessageEventArgs(retval) parameter will only be returned if the WndMessageEventArgs.ndef parameter is set to non-zero (even though it obviously cannot block default processing that has already occurred).

	The retval and nodef parameters have no effect when the subclassing type is asynchronous.
--	---

WinHook Class

This class serves as the base class for the KeyHook class and is used to set a Windows hook. The most likely *HookTypes* that you will use are the GetMessage, Mouse, Keyboard and CallWndProc hooks. Refer to the HookType property on how to use these hook types. Note that this object raises different events for different types of hooks.

Use of the nodef event parameter

Most WinHook event argument classes include a nodef field which can be set during event processing. This event provides direction to the dwshengine80.dll engine to not call the CallNextHookEx function. If other applications have placed the same hook, this will in many cases prevent the other hook from being called. If the hook accepts a True return value to indicate the message was handled, setting nodef will return True. For CBTHooks, setting nodef to True will cause the value specified by the BlockCBTOperation parameter to be returned.

Properties

AsyncNotification	<p>[VB] Property AsyncNotification As Boolean [C#] bool AsyncNotification</p> <p>True to specify that the WinHook object should trigger an event when the message is detected. False to specify that the WinHook object should trigger an event during the course of normal windows processing.</p>
Enabled	<p>[VB] Property Enabled As Boolean [C#] bool Enabled</p> <p>Enabled or disables the Windows hook.</p>
HookType	<p>[VB] Property HookType As HookTypes [C#] HookTypes HookType</p> <p>Windows provides a number of different types of windows hooks. The WinHook object supports twelve types of hooks. Two additional hook types, the keyboard hook and low level keyboard hook, are supported by the KeyHook object. Each type of hook detects a different subset of messages and different types. You will probably want to experiment to determine which hook type is appropriate for your needs.</p>

	<p>An in-depth discussion of each type of hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this documentation has access to the Windows software development kit or Developer's Network CD-ROM.</p> <p>CallWndProc – Implements a WH_CALLWNDPROC hook. This hook is triggered any time a message is sent to a window function. This hook type detects every message received by a window. Even with the advanced filtering used by SpyWorks, use of this hook can impact system performance and should be avoided if possible.</p> <p>CallWndProcRet – Implements a WH_CALLWNDPROCRET hook. This hook is triggered any time a windows function returns from a message. This hook type detects every message received by a window. Even with the advanced filtering used by SpyWorks, use of this hook can impact system performance and should be avoided if possible.</p> <p>CBT – Implements a WH_CBT hook. This hook is used to implement computer based training applications, providing information on a variety of windows events.</p> <p>ForegroundIdle – Implements a WH_FOREGROUNDIDLE hook. This hook is used to detect when the foreground thread is about to become idle.</p> <p>GetMessage – Implements a WH_GETMESSAGE hook. This hook is triggered any time an API function called GetMessage (or PeekMessage) 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.</p> <p>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.</p> <p>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.</p> <p>Keyboard – Implements a WH_KEYBOARD hook. This hook is triggered by keyboard events.</p> <p>KeyboardLL – Implements a WH_KEYBOARD_LL hook.</p>
--	---

	<p>This hook is triggered by keyboard events.</p> <p>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.</p> <p>Mouse – Implements a WH_MOUSE hook. This hook is triggered by mouse events.</p> <p>MouseLL – Implements a WH_MOUSE_LL hook. This hook is triggered by mouse events.</p> <p>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.</p> <p>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.</p>
<p>HwndParam</p>	<p>[VB] Property HwndParam As IntPtr [C#] IntPtr HwndParam</p> <p>This property only has an effect when the Monitor property is set to '5 - HwndParam', or '6 – HwndAndChildren'. When set at runtime to a window handle, only messages sent to this window, or to this window and to its descendants (depending on the Monitor property), will be detected.</p>
<p>Messages</p>	<p>[VB] Property Messages As WindowsMessageList [C#] WindowsMessageList Messages</p> <p>This property refer to a WindowsMessageList object that contains the Windows messages to detect. This property is initially set to Nothing which will hook all Windows messages. Most Windows messages are also exposed through one of the Desaware.SpyWorks.*Messages enumerator classes.</p>
<p>Monitor</p>	<p>[VB] Property Monitor As HookMonitor [C#] HookMonitor Monitor</p> <p>Windows hooks are designed to intercept messages on a global basis. The Monitor property provides a degree of filtering to help you limit which messages to detect in order to improve system efficiency. The values of this property are as follows:</p> <p>EntireSystem - Messages will be detected for all processes.</p>

	<p>HwndAndChildren - Only messages going to window whose handle is set into the HwndParam property and the children of that window will be detected.</p> <p>HwndParam - Only messages going to the window whose handle is set into the HwndParam property will be detected.</p> <p>ProcessParam - Only messages going to windows owned by the process whose process ID is set into the ProcessParam property will be detected.</p> <p>ThisProcess - Only messages going to windows in the process that creates this WinHook object will be detected.</p> <p>ThisThread - Only messages going to windows in the thread that creates this WinHook object will be detected.</p> <p>ThreadParam - Only messages going to windows owned by the thread whose thread ID is set into the ThreadParam property will be detected.</p>
ProcessParam	<p>[VB] Property ProcessParam As Integer [C#] int ProcessParam</p> <p>This property only has an effect on the Windows hook when the Monitor property is set to ProcessParam. In this case, only messages going to Windows belonging to the process specified by ProcessParam will be detected. Be sure to set this to the process id of a process.</p>
ThreadParam	<p>[VB] Property ThreadParam As Integer [C#] int ThreadParam</p> <p>This property only has an effect on the Windows hook when the Monitor property is set to ThreadParam. In this case, only messages going to Windows belonging to the thread specified by ThreadParam will be detected. Be sure to set this to the thread id of a thread.</p>

Events

OnCBTHook	<p>[VB] Sub WinHook1_OnCBTHook(ByVal sender As Object, ByVal e As Desaware.SpyWorks.CBTHookEventArgs) [C#] void WinHook1_OnCBTHook(object sender,</p>
-----------	---

`Desaware.SpyWorks.CBTHookEventArgs e)`

This event is triggered for messages detected when the `HookType` property is set to `CBT`. The `CBTHookEventArgs` parameters are as follows:

`ActivateStruct (CBTActivateStruct)` – Refer to your Windows API documentation for the `CBTProc` function for information on this parameter.

`BlockCBTOperation (Boolean)` – Set to `True` to prevent the current event from taking place. This parameter sets the return value to the `CBTProc` hook function. Refer to your Windows API documentation for the `CBTProc` function for information on this parameter and which `CBT` events it applies to. You must also set the `nodef` parameter to non-zero for this property to take effect.

`code (CBTMessageType)` – Refer to your Windows API documentation for the `CBTProc` function for information on this parameter.

`CreateWndStruct (CBTCreateStruct)` – Refer to your Windows API documentation for the `CBTProc` function for information on this parameter.

`lParam (Integer)` – This parameter depends on the `code` parameter.

`MouseHookStruct (CBTMouseHookStruct)` – Refer to your Windows API documentation for the `CBTProc` function for information on this parameter.

`MoveSizeRect (Rectangle)` – Refer to your Windows API documentation for the `CBTProc` function for information on this parameter.

`nodef (Short)` – Refer to *Use of the nodef parameter*. You must set this to non-zero for the `BlockCBTOperation` parameter to take effect.

`wParam (Integer)` – This parameter depends on the `code` parameter.

An in-depth discussion of `CBT` hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this documentation has access to the Windows software

	<p>development kit or Developer's Network CD-ROM.</p> <p>The IParam parameter is frequently a pointer to a structure (whether this is the case, and the type of structure, depends on the code). The WinHook object is aware of the types of structures supported, and ensures that the structure and contents are copied into the current memory address space - an important issue under Windows 95/98/ME and Windows NT/2000/XP where address pointers may not be valid when moved between processes.</p>
<p>OnForegroundIdleHook</p>	<pre>[VB] Sub WinHook1_OnForegroundIdleHook(ByVal sender As Object, ByVal e As Desaware.SpyWorks.ForegroundIdleHookEventArgs) [C#] void WinHook1_OnForegroundIdleHook(object sender, Desaware.SpyWorks.ForegroundIdleHookEventArgs e)</pre> <p>This event is triggered when the HookType property is set to ForegroundIdle and the foreground thread is about to become idle. The ForegroundIdleHookEventArgs parameters are as follows:</p> <p>nodef (Short) - Refer to <i>Use of the nodef parameter</i>.</p> <p>Use this hook to detect when your process is entering an idle state. At this point, you can run some background operations without taking CPU time from your main process.</p>
<p>OnJournalPlaybackHook</p>	<pre>[VB] Sub WinHook1_OnJournalPlaybackHook(ByVal sender As Object, ByVal e As Desaware.SpyWorks.JournalPlaybackHookEventArgs) [C#] void WinHook1_OnJournalPlaybackHook(object sender, Desaware.SpyWorks.JournalPlaybackHookEventArgs e)</pre> <p>This event is triggered when the HookType property is set to JournalPlayback and the messages are detected. The JournalPlaybackHookEventArgs parameters are as follows:</p> <p>code (JournalMessageType) – Refer to your Windows API documentation for the JournalPlaybackProc function for information on this parameter.</p> <p>delay (Integer) – Specifies the delay in milliseconds until the message will be placed in the system queue. Zero for no delay (default).</p>

	<p>msg (Integer) – The keyboard or mouse message to place in the system queue.</p> <p>mtime (Integer) – Specifies the time stamp for the message.</p> <p>paramH (Integer) – This parameter depends on the msg parameter.</p> <p>paramL (Integer) – This parameter depends on the msg parameter.</p> <p>wnd (IntPtr) – Specifies the window handle.</p> <p>An in-depth discussion of Journal hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this event has access to the Windows software development kit or Microsoft Developer’s Network CD-ROM.</p>
OnJournalRecordHook	<pre>[VB] Sub WinHook1_OnJournalRecordHook(ByVal sender As Object, ByVal e As Desaware.SpyWorks.JournalRecordHookEventArgs) [C#] void WinHook1_OnJournalRecordHook(object sender, Desaware.SpyWorks.JournalRecordHookEventArgs e)</pre> <p>This event is triggered when the HookType property is set to JournalRecord and the messages are detected. The JournalRecordHookEventArgs parameters are as follows:</p> <p>code (JournalMessageType) – Refer to your Windows API documentation for the JournalRecordProc function for information on this parameter.</p> <p>delay (Integer) – Specifies the delay in milliseconds until the message will be placed in the system queue. Zero for no delay (default).</p> <p>msg (Integer) – The keyboard or mouse message to place in the system queue.</p> <p>mtime (Integer) – Specifies the time stamp for the message.</p> <p>paramH (Integer) – This parameter depends on the msg parameter.</p> <p>paramL (Integer) – This parameter depends on the msg parameter.</p>

	<p>wnd (IntPtr) – Specifies the window handle.</p> <p>An in-depth discussion of Journal hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this event has access to the Windows software development kit or Microsoft Developer’s Network CD-ROM.</p>
OnMessageHook	<pre>[VB] Sub WinHook1_OnMessageHook(ByVal sender As Object, ByVal e As Desaware.SpyWorks.MessageHookEventArgs) [C#] void WinHook1_OnMessageHook(object sender, Desaware.SpyWorks.MessageHookEventArgs e)</pre> <p>This event is triggered for messages detected when the HookType property is set to either CallWndProc, CallWndProcRet, GetMessage, MessageFilter or SysMessageFilter. The MessageHookEventArgs parameters are as follows:</p> <p>handling (MessageHandling) – Valid only for the GetMessage HookType. If set to Remove, indicates that the message is removed from the message queue after exiting this event. If set to NoRemove, indicates that the message will remain on the message queue after exiting this event (and will trigger another event later).</p> <p>hwnd (IntPtr) – Window handle for the message.</p> <p>inproccall (Boolean) – For the CallWndProc and CallWndProcRet hook types, this parameter will be True if the message was sent by the executing thread (this will be the event thread unless the AsyncNotification property is True).</p> <p>lp (Integer) – This parameter depends on the msg parameter.</p> <p>msg (Integer) – Windows message that was detected.</p> <p>nodef (Short) – Refer to <i>Use of the nodef parameter. You must set this to non-zero to return a value with the CallWndProcRet hook type.</i></p> <p>process (Integer) – Process id of the process the message is intended for.</p> <p>retval (Integer) – Sets the return value from the hook, valid only when the <i>HookType</i> property is set to CallWndProcRet</p>

	<p>and the <code>nodef</code> property is set to <code>True</code>.</p> <p><code>source</code> (<code>MessageInputSource</code>) – Describes the type of input event that generated the windows message. Valid only when the <code>HookType</code> property is set to <code>MessageFilter</code> or <code>SysMessageFilter</code>.</p> <p><code>wp</code> (<code>Integer</code>) – This parameter depends on the <code>msg</code> parameter.</p> <p>An in-depth discussion of these hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this event has access to the Windows software development kit or Microsoft Developer's Network CD-ROM.</p>
<p>OnMouseHook</p>	<pre>[VB] Sub WinHook1_OnMouseHook(ByVal sender As Object, ByVal e As Desaware.SpyWorks.MouseHookEventArgs) [C#] void WinHook1_OnMouseHook(object sender, Desaware.SpyWorks.MouseHookEventArgs e)</pre> <p>This event is triggered for messages detected when the <code>HookType</code> property is set to <code>Mouse</code> or <code>MouseLL</code>. The <code>MouseHookEventArgs</code> parameters are as follows:</p> <p><code>flags</code> (<code>Integer</code>) – Valid only when the <code>HookType</code> is set to <code>MouseLL</code>. Bit field that specifies the event-injected flag. Currently, Bit 0 is set if this message was injected.</p> <p><code>handling</code> (<code>MessageHandling</code>) - If set to <code>Remove</code>, indicates that the message is removed from the message queue after exiting this event. If set to <code>NoRemove</code>, indicates that the message will remain on the message queue after exiting this event (and will trigger another event later).</p> <p><code>hitcode</code> (<code>Integer</code>) – Valid only when the <code>HookType</code> is set to <code>Mouse</code>. A 32 bit hit test code identifying the type of screen object at the position indicated. Refer to your Windows API reference or the on-line help for a list of these codes.</p> <p><code>hwnd</code> (<code>IntPtr</code>) – Valid only when the <code>HookType</code> is set to <code>Mouse</code>. Window handle for the message.</p> <p><code>mousedata</code> (<code>Integer</code>) - Valid only when the <code>HookType</code> is set to <code>MouseLL</code>. Contains information on the mouse wheel or x button.</p> <p><code>msg</code> (<code>Integer</code>) – Window message that was detected.</p>

	<p>nodef (Short) - Refer to <i>Use of the nodef parameter</i>.</p> <p>process (Integer) – Process id of the process the message is intended for.</p> <p>time (Integer) – Valid only when the HookType is set to MouseLL. Time stamp for this message.</p> <p>x (Integer) – The x location of the cursor in screen coordinates.</p> <p>xtra (Integer) – Specifies extra information associated with the message.</p> <p>y (Integer) – The y location of the cursor in screen coordinates.</p> <p>An in-depth discussion of Mouse or MouseLL hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this event has access to the Windows software development kit or Microsoft Developer’s Network CD-ROM.</p>
<p>OnShellHook</p>	<pre>[VB] Sub WinHook1_OnShellHook(ByVal sender As Object, ByVal e As Desaware.SpyWorks.ShellHookEventArgs) [C#] void WinHook1_OnShellHook(object sender, Desaware.SpyWorks.ShellHookEventArgs e)</pre> <p>This event is triggered when the WinHook object receives notifications of shell events from the system when the HookType property is set to Shell. The ShellHookEventArgs parameters are as follows:</p> <p>code (ShellMessageType) – Refer to your Windows API documentation for the ShellProc function for information on this parameter.</p> <p>commandhandled (Boolean) – If the code is of type HSHELL_APPCOMMAND, this parameter sets the return value of the ShellProc function indicating whether the command was processed. You must set the nodef parameter to True to use this as well.</p> <p>IParam (Integer) - This parameter depends on the code parameter.</p>

	<p>MinRect (Rectangle) – Contains the window size information for the minimized/maximized window, valid only when the code is set to GetMinRect.</p> <p>nodef (Short) – Refer to <i>Use of the nodef parameter</i>. You must set this to non-zero to use the commandhandled parameter.</p> <p>wParam (Integer) – This parameter depends on the code parameter.</p> <p>An in-depth discussion of Shell hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this event has access to the Windows software development kit or Microsoft Developer's Network CD-ROM.</p>
--	---

dwsbc80.ocx Reference

Introduction

The dwsbc80.ocx file is an ATL based COM/ActiveX control that exposes Windows Subclass technology. Although our testing have found that this control works fine on the Visual Studio .NET platform (with one exception – refer to the next section for details), Desaware recommends using the new Desaware.shcomponent.dll component in place of the previous Subclass and WinHook COM controls for development on the Visual Studio .NET platform.

Features:

- Detect windows messages for any window, form or control (those with a window handle) in the system and trigger an event when it occurs.
- Detect messages before the Windows default processing, after the Windows default processing, or simply post it to yourself for later examination.
- When detecting messages before the default processing, you can change the message or its parameters, or cause the message to be completely ignored.
- Subclass windows in other applications - you can effectively create .NET applications that add their own menu commands to another application so that your .NET application acts as an Add-on to that application.
- Specify exactly which messages to detect - this minimizes the overhead to provide the fastest possible performance.
- Detect registered windows messages.

- Delayed event processing - allows you to "post" an event to yourself without setting up a timer control.
- Each dwsbc80.ocx subclass control can subclass multiple windows or controls (limited only by memory).
- In order to retrieve strings or data across processes, several cross-process string and data access functions have been included.

Properties

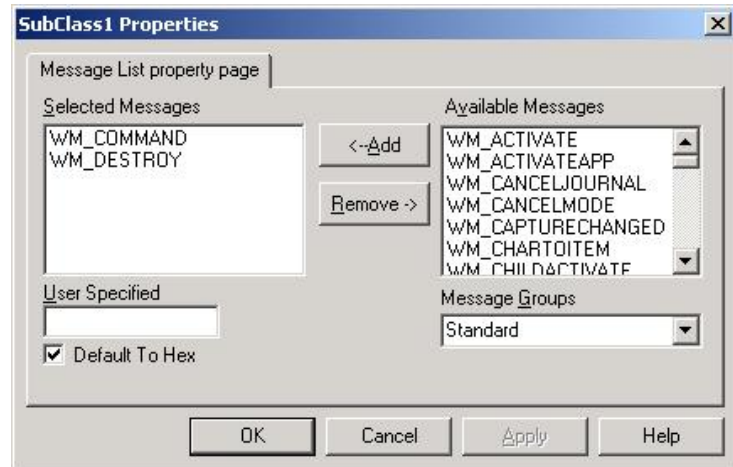
The dwsbc80.ocx subclass control includes support for both COM/ActiveX and .NET development platforms. Some properties are useful or applicable only in the COM/ActiveX development platform. Only those properties valid in the .NET development platform are described here.

AddHwnd	<p>[VB] WriteOnly Property AddHwnd As Integer [C#] int AddHwnd [set]</p> <p>Setting the AddHwnd property to the handle of a window causes the specified window to be added to the array of windows being subclassed.</p>
ClearMessage	<p>[VB] WriteOnly Property ClearMessage As Integer [C#] int ClearMessage [set]</p> <p>At runtime, setting this property to the value of a message number causes that message number to be removed from the filter list of messages that will be detected.</p>
CrossTaskTimeout	<p>[VB] Property CrossTaskTimeout As Integer [C#] int CrossTaskTimeout</p> <p>This property contains the timeout value when performing cross-process subclassing or hooking. When doing cross-process subclassing or hooking, it is quite possible for an application to freeze up the entire system. The SubClass control has a watchdog timer during subclassing that, when it times out, will interrupt the subclass. This property sets the timeout value. Be aware that processing time within the WinMessageX event counts toward the timeout time, so if there are any breakpoints in the WinMessageX event, set this property to a very high value.</p>
HookCount	<p>[VB] ReadOnly Property HookCount As Integer [C#] int HookCount [get]</p>

	<p>This property returns the total number of windows subclassed using the subclassing array.</p>
HwndArray	<p>[VB] Function get_HwndArray (ByVal index As Short) As Integer [C#] int get_HwndArray (short index)</p> <p>This property array allows you to read the window handles of the windows that are being subclassed in the subclassing array.</p>
HwndParam	<p>[VB] Property HwndParam As Integer [C#] int HwndParam</p> <p>Setting this property to the handle of a window at runtime causes the SubClass control to subclass that window. Set this property to zero to end subclassing using this property. This property has no effect on controls subclassed using the subclassing array.</p>
MessageArray	<p>[VB] Function get_MessageArray (ByVal index As Short) As Integer [C#] int get_MessageArray (short index)</p> <p>This property array can be used to read the message numbers that are currently being intercepted by the SubClass control.</p>
MessageCount	<p>[VB] ReadOnly Property MessageCount As Short [C#] short MessageCount [get]</p> <p>This property can be read to determine the number of messages that are currently being detected, not including registered messages.</p>
Messages	<p>[VB] Property Messages As Integer [C#] int Messages</p> <p>This property is used to determine which messages the subclass control will detect. It can be used in two ways. At design time, click on the '...' in the property bar to bring up the SubClass control's Message List form which is used to select messages. At runtime, setting this property to the value of a message number causes that message number to be added to the filter list of messages that will be detected. You can use the MessageArray and MessageCount</p>

properties to determine which messages have been set for a SubClass control.

Only messages that are specified will be detected. If no messages are specified, the SubClass control will detect all messages.



Refer to the RegMessage properties for information on detecting registered windows messages.

The Message List form appears when you press the '...' button on the property window for the Message property. Messages are divided into groups as defined in the SpyWorks.ini file. You can use the Message Groups combo box to select the group from which to select messages.

The available messages for each group appears in the Available Messages list box. You can select a message by clicking on the Add button when the message is highlighted, or double clicking on the message.

The Remove button can be used to cancel detection of a message.

If a message is not already defined by the system, you have two choices. You can add the message to the SpyWorks.ini file, or enter the message value directly into the user defined edit box. This edit box accepts the standard &H or 0X format to specify hexadecimal notation, or you can leave the "Default Hex" check box checked, in which case your entry is assumed to be always in hex.

In some cases a message number is used by multiple groups. In this case, the Message Select dialog box will use the most recently selected group to determine the name of the message. Messages are saved internally by

	value, not name.
Persist	<p>[VB] Property Persist As Boolean [C#] bool Persist</p> <p>Some OLE controls have a window handle only when active; not at other times. If the OLE control being subclassed destroys and recreates its window, the SubClass control will lose the subclass link with the OLE control. When the Persist property is True, the SubClass control will monitor for the destruction and recreation of the window that is being subclassed, and will automatically re-subclass the OLE control when the window is recreated.</p>
PostEvent	<p>[VB] WriteOnly Property PostEvent As Integer [C#] int PostEvent [set]</p> <p>Sometimes you will run into a situation where you want to do something "later", but don't want to go through the hassle of setting up a timer control (not to mention dealing with the delay inherent in setting a timer delay). The PostEvent property can be used to place an event in the message processing queue that will occur during the course of normal event processing. The long value set into this property will be passed as a parameter to the DelayedEvent event.</p> <p>This property is especially valuable to divide the processing of a message into two parts - the part that needs to be processed immediately with the message (at which time there may be limits on the allowed operations), and the part of processing that can be deferred.</p>
RegMessage1 to RegMessage5	<p>[VB] Property RegMessage1 As String [C#] string RegMessage1</p> <p>Most messages dealt with in Windows are specified by constant values. In some cases, however, messages are known by name and their value can change each time the application is run. These are known as registered messages.</p> <p>Each SubClass control can detect up to five registered messages. Simply set the contents of the RegMessage1 through RegMessage5 properties to the message name. This property can be set at either runtime or design time.</p>

RegMessageNum	<p>[VB] Function get_RegMessageNum (ByVal index As Short) As Integer [C#] int get_RegMessageNum (short index)</p> <p>This array gives the message numbers associated with the registered messages specified in the RegMessage properties. The RegMessage properties are specified by a string containing the name of the registered message to hook, while the RegMessageNum property array will give the actual message number that is associated with the registered message.</p>
RemoveHwnd	<p>[VB] WriteOnly Property RemoveHwnd As Integer [C#] int RemoveHwnd [set]</p> <p>Setting this property to the handle of a window removes that window from the subclassing array, and stops subclassing of that window.</p>
Type	<p>[VB] Property Type As Desaware.SpyWorks.dwsbcNET.enumTypeConstants [C#] Desaware.SpyWorks.dwsbcNET.enumTypeConstants Type</p> <p>There are three types of subclassing possible:</p> <p>0 = Pre-Default (sbcPreDefault) - The message is intercepted before default processing for the message takes place. Any modifications you make to the message number or wp and lp parameters for the message will be passed on to the default message handler. You also have the option to prevent default message processing and specify a return value to the calling function.</p> <p>1 = Post-Default (sbcPostDefault) - The message is intercepted after default processing for the message takes place. Changes to the message number or wp and lp parameters will have no effect on the default message processing, but would affect any other SubClass controls that are subclassing this window.</p> <p>2 = Posted (sbcPosted) - Any time the message is detected, the message and parameters are posted to the SubClass control. Changes to the message number or wp and lp will have no effect. The WndMessage event will occur at some time after the message has been processed, during the course of normal event processing.</p>

	<p>This method is ideal for detecting messages that do not have to be handled immediately. There are no restrictions on the contents of the event procedure when this technique is used.</p> <p>In cases where you need to perform more than one type of subclassing on a window, you will need to use multiple SubClass controls and set each control's Type property to a different type.</p>
UseOnlyXEvent	<pre>[VB] Property UseOnlyXEvent As Boolean [C#] bool UseOnlyXEvent</pre> <p>When this property is True, all subclass events will be forced to the WndMessageX event, including messages that would normally go to the WndMessage event. When this property is False, messages are sent to the appropriate event.</p>

Methods

The dwsbc80.ocx SubClass control includes support for both COM/ActiveX and .NET development platforms. Some methods are useful or applicable only in the COM/ActiveX development platform. Only those methods valid in the .NET development platform are described here.

GetAnsiString	<pre>[VB] Function GetAnsiString (ByVal Address As Integer, ByVal Process As Integer) As String [C#] string GetAnsiString (int Address, int Process)</pre> <p>This method takes an Ansi string address and the process ID, and returns the actual string. This allows the ability to retrieve a string from cross-task processes.</p>
GetUnicodeString	<pre>[VB] Function GetUnicodeString (ByVal Address As Integer, ByVal Process As Integer) As String [C#] string GetUnicodeString (int Address, int Process)</pre> <p>This method takes a Unicode string address and the process ID, and returns the actual string. This allows the ability to retrieve a string from cross-task processes.</p>

Events

<p>DelayedEvent</p>	<pre>[VB] Sub SubClass1_DelayedEvent(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_DelayedEventEvent) Handles SubClass1.DelayedEvent [C#] void SubClass1_DelayedEvent(object sender, AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_DelayedEventEvent e)</pre> <p>The PostEvent property is used to trigger this event. It is typically used to delay some operation without going through the trouble of setting up a timer. You may perform any operation during this event. The DelayedEventEvent field is described as follows:</p> <p>Ivalue (Integer) – Value set into the PostEvent property that triggered this event.</p>
<p>WndMessage</p>	<pre>[VB] Sub SubClass1_WndMessage(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageEvent) Handles SubClass1.WndMessage [C#] void SubClass1_WndMessage(object sender, AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageEvent e)</pre> <p>This event is triggered for subclassed controls and forms that are in the same process as the SubClass control. If, however, the UseOnlyXEvent property is True, this event will NOT be triggered for any subclassed controls and forms. The WndMessageEvent fields are as follows:</p> <p>hwnd (Integer) – The window handle.</p> <p>msg (Integer) – The message number.</p> <p>wp (Integer) – The wParam parameter.</p> <p>lp (Integer) – The lParam parameter.</p> <p>retval (Integer) – The 32 bit value to return to the calling function.</p> <p>nodef (Short) – Set to True to prevent default message processing.</p> <p>If the Type property is set to '0 - Pre-Default' (sbcPreDefault), the retval parameter only returns a value to the calling function if you set the nodef parameter to non-zero. If you leave the nodef parameter as zero, the default window function for the window is called and the value that it returns is passed on to the calling function.</p>

	<p>When the Type property is set to '1 - Post-Default' (sbcPostDefault), the retval parameter contains the value returned by the default window function for the window. You can override this return value by changing the value of the retval parameter and setting the nodef parameter to non-zero. Remember, the retval parameter will only be returned if the nodef parameter is set to non-zero (even though it obviously cannot block default processing that has already occurred).</p>
<p>WndMessageX</p>	<pre>[VB] Sub SubClass1_WndMessageX(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageXEvent) Handles SubClass1.WndMessageX [C#] void SubClass1_WndMessageX(object sender, AxDesaware.SpyWorks.dwsbcNET._DDwsbcEvents_WndMessageXEvent e)</pre> <p>This event is triggered for subclassed controls and forms that are NOT in the same process as the SubClass control. If, however, the UseOnlyXEvent property is True, this event will be triggered for all subclassed controls and forms. The WndMessageXEvent fields are as follows:</p> <p>wnd (Integer) – The window handle.</p> <p>msg (Integer) – The message number.</p> <p>wp (Integer) – The wParam parameter.</p> <p>lp (Integer) – The lParam parameter.</p> <p>retval (Integer) – The 32 bit value to return to the calling function.</p> <p>nodef (Short) – Set to True to prevent default message processing.</p> <p>process (Integer) – The process ID of the message sender.</p> <p>If the Type property is set to '0 - Pre-Default' (sbcPreDefault), the retval parameter only returns a value to the calling function if you set the nodef parameter to non-zero. If you leave the nodef parameter as zero, the default window function for the window is called and the value that it returns is passed on to the calling function.</p> <p>When the Type property is set to '1 - Post-Default' (sbcPostDefault), the retval parameter contains the value returned by the default window function for the window. You can override this return value by changing the value of the retval parameter and setting the nodef parameter to non-zero. Remember, the retval parameter will only be returned if the nodef parameter is set to non-zero (even though it obviously cannot block default processing that has already occurred).</p>

dwshk80.ocx Reference

Introduction

The dwshk80.ocx file is an ATL based COM/ActiveX control that exposes Windows hook technology. The documentation for the dwshk80.ocx control will be split into the keyboard hook control section and a windows hook control section. Although our testing has found that this control works fine on the Visual Studio .NET platform, Desaware recommends using the new Desaware.shcomponent.dll component in place of the previous Subclass and WinHook COM controls for development on the Visual Studio .NET platform.

Keyboard hook features:

- Implementation of the WH_KEYBOARD and WH_KEYBOARD_LL hook types.
- Detect and disable special keys. Ability to detect Ctrl+Alt+Del, ability to disable Ctrl+Esc, Alt+Tab, Alt+Esc, and many other special keys.
- Receive all keystrokes sent to any process. Keystrokes are detected before they are sent to the application; thus you can even trap special keys such as the enter, control break, and tab keys.
- Place a system wide keyboard hook. This allows your .NET application to be a "hotkey" type application that is triggered by a specific key sequence regardless of which application is active.

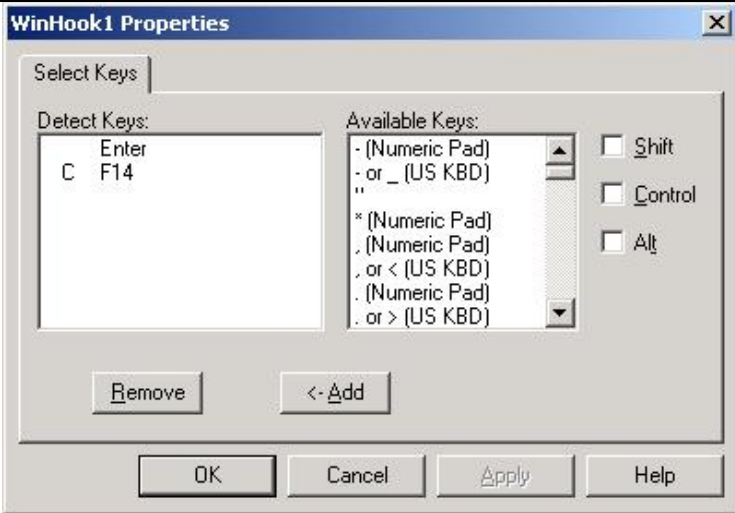
Keyboard hook Properties

The dwshk80.ocx keyboard hook control includes support for both COM/ActiveX and .NET development platforms. Some properties are useful or applicable only in the COM/ActiveX development platform. Only those properties valid in the .NET development platform are described here.

KeyArray	<code>[VB] ReadOnly Property KeyArray (ByVal KeyIndex As Short) As Integer</code> <code>[C#] int get_KeyArray (short KeyIndex)</code> This property array can be used to read the 32 bit key values for keys that are currently being intercepted by the KeyBoard hook control. The key value is specified in the KeyBoard hook control key value format.
KeyboardEvent	<code>[VB] Property KeyboardEvent As Desaware.SpyWorks.dwshkNET.KeyEventConstants</code>

	<p>[C#] Desaware.SpyWorks.dwshkNET.KeyEventConstants KeyboardEvent</p> <p>Newer versions of this control divide the old KbdHook event into two separate events: one that detects key up events, the other that detects key down events. These events are easier to use and should prove popular for new applications.</p> <p>0 - Extended Events (shkKbdExtended) – Enable the two new events: KeyDownHook and KeyUpHook.</p> <p>1 - Use KbdHook (shkUseKbdHook) – Use the old KbdHook event only.</p> <p>2 - Use low level Hook (shkUseKbdLLHook) – Use the low level keyboard hook.</p>
KeyboardHook	<p>[VB] Property KeyboardHook As Desaware.SpyWorks.dwshkNET.KeyboardHookConstants [C#] Desaware.SpyWorks.dwshkNET.KeyboardHookConstants KeyboardHook</p> <p>Enables and disables keyboard hooking by this control, and determines the scope of the keyboard hook:</p> <p>0 – Disabled (shkKbdDisabled) – Disable the keyboard hook for this instance of the control.</p> <p>1 - This Task (shkKbdThisTask) – Only hook keyboard messages coming from the process of which the control is a part.</p> <p>2 - Entire System (shkKbdEntireSystem) – Hook all keyboard messages from the entire system.</p> <p>3 – TaskParam (shkKbdTaskParam) – Hook all keyboard messages from the process specified in the TaskParam property. It is not possible to have a different task ID for the keyboard hook and the windows hook; if a different task ID is necessary, use two instances of this control.</p> <p>4 – ThreadParam (shkKbdThreadParam) – Hook all keyboard messages from the thread specified in the ThreadParam property. It is not possible to have a different thread ID for the keyboard hook and the windows hook; if</p>

	<p>a different thread ID is necessary, use two instances of this control.</p>
KeyCount	<p>[VB] ReadOnly Property KeyCount As Short [C#] short KeyCount [get]</p> <p>This property can be read to determine the number of keys that are currently being detected.</p>
KeyIgnoreCapsLock	<p>[VB] Property KeyIgnoreCapsLock As Boolean [C#] bool KeyIgnoreCapsLock</p> <p>When this property has a value of False, alphabetic characters are shifted according to the state of the CapsLock key. Other keys are not affected. When this property has a value of True, the CapsLock key has no effect.</p>
Keys	<p>[VB] Property Keys As Integer [C#] int Keys</p> <p>This 32 bit property is used to determine which keys the KeyBoard hook control will detect. It can be used in two ways. At design time, click on the '...' in the property bar to bring up the Key Select Window which is used to select keys.</p> <p>At runtime, setting this property to a key value causes that key to be added to the filter list of keys that will be detected. This value is in the KeyBoard hook control's key value format specified earlier in this chapter.</p> <p>If you do not specify a filter list for keys, all keys will be detected.</p> <p>The Key Select Window is shown below.</p>

	 <p>This window is used to select keys to be intercepted by this control. The Available list box displays a list of all virtual keys. Those currently being detected will be listed in the Detect list box.</p> <p>Entries in the Detect list box may be preceded by one or more of the letter 'S', 'C' and 'A' indicating the state of the Shift, Ctrl and Alt keys that are required for detection.</p> <p>The Add button will add a key to the Detect list box based on the virtual key selected in the Available list box and the settings of the Shift, Control and Alt checkboxes.</p> <p>You can remove any key from the Detect list box using the Remove control.</p> <p>Entries in both the Detect and Available list boxes are sorted in alphabetical order.</p>
KeyViewPeeked	<pre>[VB] Property KeyViewPeeked As Boolean [C#] bool KeyViewPeeked</pre> <p>Keyboard hook events are triggered any time the system reads a key event from the system queue. However, reading an event does not mean that the event is always removed from the system queue. In some cases a key event is “peeked” - previewed, and it remains in the queue. This property determines whether you want to see these peeked keys. Normally, you will want to leave this property False.</p>
KeyboardNotify	<pre>[VB] Property KeyboardNotify As Desaware.SpyWorks.dwshkNET.NotifyConstants [C#] Desaware.SpyWorks.dwshkNET.NotifyConstants KeyboardNotify</pre> <p>This property determines when the KeyBoard hook event is</p>

	<p>triggered.</p> <p>0 - When Hooked (shkWhenHooked) – The KeyBoard hook event is triggered as soon as the keystroke occurs. In this case you have the option of discarding the character so that it will not be seen by the system. When this type of hook is in effect, you should abide by the same limitations that apply when subclassing windows messages - minimize the amount of code in the event and do not change focus, load or unload controls or load or unload applications.</p> <p>1 – Posted (shkPosted) – The keystroke event is posted so that the KeyBoard hook event will be triggered during the course of normal windows processing. This method is ideal for detecting the occurrence of keyboard events when immediate processing is not necessary.</p>
TaskParam	<p>[VB] Property TaskParam As Integer [C#] int TaskParam</p> <p>This 32 bit property only has an effect on the keyboard hook when the KeyboardHook property is set to '3 - TaskParam'. In this case, only keyboard messages going to Windows belonging to the process specified by TaskParam will be detected. Be sure to use a task/process handle, not an instance handle, to set this property.</p>
ThreadParam	<p>[VB] Property ThreadParam As Integer [C#] int ThreadParam</p> <p>This 32 bit property only has an effect on the keyboard hook when the KeyboardHook property is set to '4 - ThreadParam'. In this case, only keyboard messages going to Windows belonging to the thread specified by ThreadParam will be detected.</p>

Keyboard hook Methods

ClearKey	<p>[VB] Sub ClearKey (ByVal KeyVal As Integer) [C#] void ClearKey (int KeyVal)</p> <p>At runtime, calling this method with the key value of a key causes that key to be removed from the filter list of keys that will be detected. The key value is specified in the Keyboard Hook key value format.</p>
----------	---

Keyboard hook Events

<p>KbdHook</p>	<pre>[VB] WinHook1_KbdHook(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KbdHookEvent) Handles WinHook1.KbdHook [C#] WinHook1_KbdHook(object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KbdHookEvent e)</pre> <p>This event occurs when a keystroke event occurs and the KeyboardEvent property is set to '1 – Use KbdHook'. The KbdHookEvent fields are as follows:</p> <p>keycode (Integer) – specifies the virtual-key code of the key that generated the keystroke message.</p> <p>keystate (Integer) – defined as follows:</p> <p>Bit 0 – 15 number of repetitions of this key Bit 16 - 23 hardware dependent scan code. Bit 24 = 1 if this is an extended key (function key or on the numeric keypad) Bit 29 = 1 if the ALT key is down Bit 30 = 1 if the key was previously down, 0 if it was up Bit 31 = 1 if the key is being released, 0 if pressed</p> <p>shiftstate (Short) – this bit field corresponds to the modifier key as follows:</p> <p>Bit 0 = 1 The shift key is down. Bit 1 = 1 The control key is down. Bit 2 = 1 The alt key is down</p> <p>discard (Short) – If the KeyboardNotify property is set to '0 - When Hooked', setting this field to non-zero will cause the keystroke to be discarded before it is processed by Windows.</p>
<p>KeyDownHook</p>	<pre>[VB] WinHook1_KeyDownHook(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyDownHookEvent) Handles WinHook1.KeyDownHook [C#] WinHook1_KeyDownHook(object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyDownHookEvent e)</pre> <p>This event occurs when a keyboard event occurs in which the key is pressed and the KeyboardEvent property is set to '0 – Extended Events'. The KeyDownHookEvent fields are as follows:</p> <p>keycode (Integer) – specifies the virtual-key code of the key that generated the keystroke message.</p>

	<p>keystate (Short) – defined as follows:</p> <p>Bit 0 - 7 the hardware dependent scan code. Bit 8 = 1 if this is an extended key (function key or on the numeric keypad) Bit 9 = 1 if this key is being “peeked” (i.e. this event was not removed from the system queue). This is only possible if the KeyViewPeeked property is set to True. Bit 13 = 1 if the ALT key is down Bit 14 = 1 if the key was previously down, 0 if it was up. Bit 15 = 1 if the key is being released, 0 if pressed.</p> <p>shiftstate (Short) – this bit field corresponds to the modifier key as follows:</p> <p>Bit 0 = 1 The shift key is down. Bit 1 = 1 The control key is down. Bit 2 = 1 The alt key is down</p> <p>discard (Short) – If the KeyboardNotify property is set to '0 - When Hooked', setting this field to non-zero will cause the keystroke to be discarded before it is processed by Windows.</p> <p>repetitions (Short) – this field is the repeat count for this key event.</p> <p>processid (Integer) – this field is the process id of the process receiving the keys or zero to indicate the current process.</p>
KeyUpHook	<pre>[VB] WinHook1_KeyUpHook(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyUpHookEvent)Handles WinHook1.KeyUpHook [C#] WinHook1_KeyUpHook (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyUpHookEvent e)</pre> <p>This event occurs when a keyboard event occurs in which the key is released and the KeyboardEvent property is set to '0 – Extended Events'. The KeyUpHookEvent fields are identical to those of the KeyDownHook event.</p>
KeyDownHookLL	<pre>[VB] WinHook1_KeyDownHookLL (ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyDownHookLLEvent) Handles WinHook1.KeyDownHookLL [C#] WinHook1_KeyDownHookLL(object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyDownHookLLEvent e)</pre> <p>This event occurs when a keyboard event occurs in which the key is pressed and the KeyboardEvent property is set to '2 - Use low level Hook'. The KeyDownHookLLEvent fields are as follows:</p> <p>keymessage (Integer) – specifies the identifier of the keyboard message - one of the following KeyboardMessages fields: WM_KEYDOWN, WM_KEYUP,</p>

	<p>WM_SYSKEYDOWN, or WM_SYSKEYUP</p> <p>keycode (Integer) – specifies the virtual-key code of the key that generated the keystroke message.</p> <p>scancode (Integer) – Specifies a hardware scan code for the key that generated the keystroke message.</p> <p>flags (Integer) – this field is defined as follows:</p> <p>Bit 0 = 1 if the key is an extended key, such as a function key or a key on the numeric keypad. The IsExtended function can be used to test this bit.</p> <p>Bit 4 = 1 if the key was injected. The IsInjected function can be used to test this bit.</p> <p>Bit 5 = 1 if the ALT key is pressed. The IsAltPressed function can be used to test this bit. NOTE that from our testing, this bit does not seem to reflect the state of the ALT key accurately some of the time. We suggest that you use the shiftstate field instead to retrieve the state of the ALT key.</p> <p>Bit 7 = 1 if the key is being released, 0 if pressed. The IsKeyRelease function can be used to test this bit.</p> <p>time (Integer) – Specifies the time stamp for this message.</p> <p>shiftstate (Short) – this bit field corresponds to the modifier key as follows:</p> <p>Bit 0 = 1 The shift key is down.</p> <p>Bit 1 = 1 The control key is down.</p> <p>Bit 2 = 1 The alt key is down.</p> <p>processid (Integer) – this field is the process id of the process receiving the keys or zero to indicate the current process.</p> <p>discard (Short) – If the KeyboardNotify property is set to '0 - When Hooked', setting this field to non-zero will cause the keystroke to be discarded before it is processed by Windows.</p> <p>ExtraInfo (Integer) – Specifies extra information associated with this key.</p>
KeyUpHookLL	<p>[VB] WinHook1_KeyUpHookLL (ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyUpHookLLEvent) Handles WinHook1.KeyUpHookLL</p> <p>[C#] WinHook1_KeyUpHookLL(object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_KeyUpHookLLEvent e)</p> <p>This event occurs when a keyboard event occurs in which the key is released and the KeyboardEvent property is set to '2 - Use low level Hook'. The KeyUpHookLLEvent fields are identical to those of the</p>

	KeyDownHookLLEvent.
--	---------------------

Windows hook features:

- Implementation of Windows hooks including WH_MOUSE, WH_MOUSE_LL, WH_GETMESSAGE, WH_MSGFILTER, WH_SYSMSGFILTER, WH_JOURNALRECORD, WH_JOURNALPLAYBACK, WH_CBT, WH_CALLWNDPROC, WH_CALLWNDPROCRET, WH_SHELL, and WH_FOREGROUNDIDLE. Allows interception of messages going to many controls without subclassing each one.
- Full control over scope of message detection - per form, per process, per thread, or systemwide.
- Specify exactly which messages to detect - this minimizes overhead to provide the fastest possible performance.
- Detect registered windows messages.
- Detect messages as they occur, or post them for later processing.
- Ability to change or discard messages (depending on the hook and message).
- JournalPlayback allows simulation of mouse or keyboard activity.

Windows hook Properties

The dwshk80.ocx Windows hook control includes support for both COM/ActiveX and .NET development platforms. Some properties are useful or applicable only in the COM/ActiveX development platform. Only those properties valid in the .NET development platform are described here.

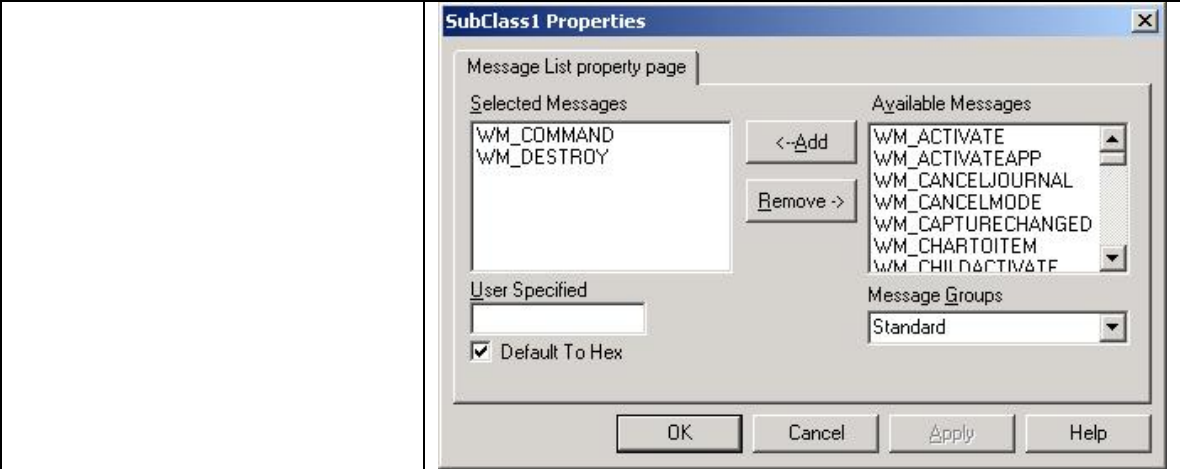
ClearMessage	<pre>[VB] WriteOnly Property ClearMessage As Integer [C#] int ClearMessage [set]</pre> <p>At runtime, setting this property to the value of a message number causes that message number to be removed from the filter list of messages that will be detected.</p>
CrossTaskTimeout	<pre>[VB] Property CrossTaskTimeout As Integer [C#] int CrossTaskTimeout</pre> <p>This property contains the timeout value when performing cross-process subclassing or hooking. When doing cross-process subclassing or hooking, it is quite possible for an application to freeze up the entire system. The Windows hook control has a watchdog timer during hooking that, when it times out, will interrupt the hooking. This property sets the timeout value. Be aware that processing time within the</p>

	<p>Windows hook events count toward the timeout time, so if there are any breakpoints in the hook events, set this property to a very high value.</p>
CurrentProcessFlag	<p>[VB] ReadOnly Property CurrentProcessFlag As Integer [C#] int CurrentProcessFlag [get]</p> <p>This property has several different meanings, depending upon which message hook is being processed. During the handling of the WH_CALLWNDPROC and WH_CALLWNDPROCRET message hooks, this property can be read to determine if the message was intercepted from another process or not: the property will be non-zero if the message was intercepted from another process.</p> <p>During the handling of the WH_GETMESSAGE message hooks, the CurrentProcessFlag property can be read to determine if the message was actually removed from the queue, or just peeked without removal. If this property is set to PM_REMOVE, then the message was actually removed from the queue; if this property is set to PM_NOREMOVE, then the message was peeked but not removed from the queue.</p>
HookEnabled	<p>[VB] Property HookEnabled As Boolean [C#] bool HookEnabled</p> <p>Enables or disables the Windows hook. Has no effect on the keyboard hook.</p>
HookType	<p>[VB] Property HookType As Desaware.SpyWorks.dwshkNET.HookTypeConstants [C#] Desaware.SpyWorks.dwshkNET.HookTypeConstants HookType</p> <p>Windows provides a number of different types of windows hooks. The dwshk80.ocx control supports thirteen types of hooks. Two types, the keyboard hook and low-level keyboard hook, are supported by the keyboard hook section of the dwshk80.ocx file. Each type of hook detects a different subset of messages and different types. You will probably want to experiment to determine which hook type is appropriate for your needs.</p> <p>Keep in mind that different hook types cause different events to be triggered - be sure you use the correct event</p>

	<p>for each hook type! The type of hook to use is determined by the HookType property as follows:</p> <p>0 - WH_GETMESSAGE (shkGetMessage) – 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. The WndMessage event is triggered when a message is detected by this hook.</p> <p>1 - WH_MOUSE (shkMouse) – Implements a WH_MOUSE hook. This hook is triggered by mouse events. The MouseProc event is triggered when a mouse message is detected by this hook.</p> <p>2 - WH_MESSAGEFILTER (shkMessageProc) – 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. The MessageProc event is triggered when a message is detected by this hook.</p> <p>3 - WH_SYSMESSAGEFILTER (shkSysMessageProc) – Implements a WH_SYSMSGFILTER hook. This hook is triggered any time a system message is sent to a dialog box, message box or menu. The MessageProc event is triggered when a message is detected by this hook.</p> <p>4 - WH_CALLWNDPROC (shkWindowProc) – 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 except for internal Visual Basic messages. Even with the advanced filtering used by SpyWorks, use of this hook can impact system performance and should be avoided if possible. The WndMessage event is triggered when a message is detected by this hook.</p> <p>5 - WH_CBT (shkCBTProc) – Implements a WH_CBT hook. This hook is used to implement computer based training applications, providing information on a variety of windows events. The CBTProc event is triggered</p>
--	--

	<p>when a message is detected by this hook.</p> <p>6 - WH_JOURNALPLAYBACK (shkJournalPlayback) – 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. The JournalPlayProc event is triggered when a message is detected by this hook.</p> <p>7 - WH_JOURNALRECORD (shkJournalRecord) – Implements a WH_JOURNALRECORD hook. This hook is used to record keyboard and mouse events on the system, typically to implement a macro recorder. The JournalRecordProc event is triggered when a message is detected by this hook.</p> <p>8 - WH_SHELL (shkShell) – 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.. The ShellProc event is triggered when a message is detected by this hook.</p> <p>9 - WH_CALLWNDPROCRET (shkCallWndProcRet) – Implements a WH_CALLWNDPROCRET hook. This hook is triggered any time a window function returns from a message. This hook type does not detect every windows message except for internal Visual Basic messages. 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 currently only supported under Windows 95. The WndMessageRet event is triggered when a message is detected by this hook.</p> <p>10 - WH_MOUSE_LL (shkMouseLL) – Implements a WH_MOUSE_LL hook. This hook is triggered by mouse events. The MouseProcLL event is triggered when a mouse message is detected by this hook.</p> <p>11 - WH_FOREGROUNDIDLE (shkForegroundIdle) – Implements a WH_FOREGROUNDIDLE hook. This hook is used to detect when the foreground thread is about to become idle. The ForegroundIdleProc event is triggered when a message is detected by this hook.</p>
--	--

HwndParam	<pre>[VB] Property HwndParam As Integer [C#] int HwndParam</pre> <p>This property only has an effect when the Monitor property is set to '2 - HwndParam', or '3 - HwndParam's Kids'. When set at runtime to a window handle, only messages sent to this window and its descendants, or just to its descendants (depending on the Monitor property), will be detected.</p>
MessageArray	<pre>[VB] Function get_MessageArray (ByVal index As Short) As Integer [C#] int get_MessageArray (short index)</pre> <p>This property array can be used to read the message numbers that are currently being intercepted by the Windows hook control.</p>
MessageCount	<pre>[VB] ReadOnly Property MessageCount As Short [C#] short MessageCount [get]</pre> <p>This property can be read to determine the number of messages that are currently being detected, not including registered messages.</p>
Messages	<pre>[VB] Property Messages As Integer [C#] int Messages</pre> <p>This property is used to determine which messages the Windows hook control will detect. It can be used in two ways. At design time, click on the '...' in the property bar to bring up the Windows hook control's Message List form which is used to select messages. At runtime, setting this property to the value of a message number causes that message number to be added to the filter list of messages that will be detected. You can use the MessageArray and MessageCount properties to determine which messages have been set for a Windows hook control. Only messages that are specified will be detected. If no messages are specified, the Windows hook control will detect all messages.</p>



Refer to the RegMessage properties for information on detecting registered windows messages.

The Message List form appears when you press the '...' button on the property window for the Message property. Messages are divided into groups as defined in the SpyWorks.ini file. You can use the Message Groups combo box to select the group from which to select messages.

The available messages for each group appears in the Available Messages list box. You can select a message by clicking on the Add button when the message is highlighted, or double clicking on the message.

The Remove button can be used to cancel detection of a message.

If a message is not already defined by the system, you have two choices. You can add the message to the SpyWorks.ini file, or enter the message value directly into the user defined edit box. This edit box accepts the standard &H or 0X format to specify hexadecimal notation, or you can leave the "Default Hex" check box checked, in which case your entry is assumed to be always in hex.

In some cases a message number is used by multiple groups. In this case, the Message Select dialog box will use the most recently selected group to determine the name of the message. Messages are saved internally by value, not name.

Monitor

```
[VB] Property Monitor As
Desaware.SpyWorks.dwshkNET.MonitorConstants
[C#]
Desaware.SpyWorks.dwshkNET.MonitorConstants
Monitor
```

	<p>Windows hooks are designed to intercept messages on a global basis. The Monitor property provides a degree of filtering to help you limit which messages to detect in order to improve system efficiency. The values of this property are as follows:</p> <p>0 - This Form (shkThisForm) – Only messages going to the form that contains the Windows hook control will be detected. Messages will be detected for the form, and for all controls on the form (except, of course, for Graphical controls which are not compatible with Windows hooks).</p> <p>1 - My Siblings (shkMySiblings) – Only messages going to child controls of the form that contains the Windows hook control will be detected. Messages will not be detected for the form itself. Messages will not be detected for child controls that are Graphical controls as they are not compatible with Windows hooks.</p> <p>2 – HwndParam (shkHwndParam) – Only messages going to the form whose window handle is set into the HwndParam property will be detected. Messages will be detected for the window, and for all child windows and controls on the window (except, of course, for Graphical controls which are not compatible with Windows hooks).</p> <p>3 - HwndParam's Kids (shkHwndKids) – Only messages going to children of the window whose handle is set into the HwndParam property will be detected. Messages will not be detected for the window itself. Messages will not be detected for child controls that are Graphical controls as they are not compatible with Windows hooks.</p> <p>4 - This Task (shkThisTask) – Only messages going to windows in the process that owns this Windows hook control will be detected.</p> <p>5 – TaskParam (shkTaskParam) – Only messages going to windows owned by the process whose process ID is set into the TaskParam property will be detected. It is not possible to have a different task ID for the keyboard hook and the windows hook; if a different task ID is</p>
--	---

	<p>necessary, use two instances of this control.</p> <p>6 - Entire System (shkEntireSystem) – Messages will be detected for all processes.</p> <p>7 - This Thread (shkThisThread) – Only messages going to windows in the thread that owns this Windows hook control will be detected.</p> <p>8 – ThreadParam (shkThreadParam) – Only messages going to windows owned by the thread whose thread ID is set into the ThreadParam property will be detected. It is not possible to have a different thread ID for the keyboard hook and the windows hook; if a different thread ID is necessary, use two instances of this control.</p>
Notify	<pre>[VB] Property Notify As Desaware.SpyWorks.dwshkNET.NotifyConstants [C#] Desaware.SpyWorks.dwshkNET.NotifyConstants Notify [set]</pre> <p>This property determines when a message hook event is triggered. The values of this property are as follows:</p> <p>0 - When Hooked (shkWhenHooked) – The WndMessage, MessageProc and MouseProc events are triggered as soon as a message is detected. In this case you have the option of discarding the message so that it will not be seen by the system. When this type of hook is in effect, you should abide by the restrictions described in the section <i>Cautions on Using Subclassing</i> in the dwsbc80.ocx control description.</p> <p>1 – Posted (shkPosted) – The event is posted so that the WndMessage, MessageProc and MouseProc events will be triggered during the course of normal windows processing. This method is ideal for detecting the occurrence of messages when immediate processing is not necessary.</p>
PostEvent	<pre>[VB] WriteOnly Property PostEvent As Integer [C#] int PostEvent [set]</pre> <p>Sometimes you will run into a situation where you want to do something "later", but don't want to go through the hassle of setting up a timer control (not to mention dealing with the delay inherent in setting a timer delay).</p>

	<p>The PostEvent property can be used to place an event in the message processing queue that will occur during the course of normal event processing. The long value set into this property will be passed as a parameter to the DelayedEvent event.</p> <p>This property is especially valuable to divide the processing of a message into two parts - the part that needs to be processed immediately with the message (at which time there may be limits on the allowed operations), and the part of processing that can be deferred.</p>
<p>RegMessage1 to RegMessage5</p>	<p>[VB] Property RegMessage1 As String [C#] string RegMessage1</p> <p>Most messages dealt with in Windows are specified by constant values. In some cases, however, messages are known by name and their value can change each time the application is run. These are known as registered messages.</p> <p>Each Windows hook control can detect up to five registered messages. Simply set the contents of the RegMessage1 through RegMessage5 properties to the message name. This property can be set at either runtime or design time.</p>
<p>RegMessageNum</p>	<p>[VB] Function get_RegMessageNum (ByVal index As Short) As Integer [C#] int get_RegMessageNum (short index)</p> <p>This array gives the message numbers associated with the registered messages specified in the RegMessage properties. The RegMessage properties are specified by a string containing the name of the registered message to hook, while the RegMessageNum property array will give the actual message number that is associated with the registered message.</p>
<p>TaskParam</p>	<p>[VB] Property TaskParam As Integer [C#] int TaskParam</p> <p>This property only has an effect on the Windows hook when the Monitor property is set to '5 - TaskParam'. In this case, only messages going to Windows belonging to the process specified by TaskParam will be detected. This property is set at runtime only. Be sure to use a process handle, not an instance handle, to set this property.</p>

ThreadParam	<p>[VB] Property ThreadParam As Integer [C#] int ThreadParam</p> <p>This property only has an effect on the Windows hook when the Monitor property is set to '8 - ThreadParam'. In this case, only messages going to Windows belonging to the thread specified by ThreadParam will be detected.</p>

Windows hook Events

You should be careful of what code you place within these events - especially when the Notify property is set to '0 - When Hooked'. Follow the information in the *Cautions on Using Subclassing* section of the dwsbc80.ocx control description.

CBTProc	<p>[VB] Sub WinHook1_CBTProc(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_CBTProcEvent) Handles WinHook1.CBTProc [C#] void WinHook1_CBTProc(object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_CBTProcEvent e)</p> <p>This event is triggered for messages detected when the Monitor property is set to '5 - CBTProc'. The CBTProcEvent fields are as follows:</p> <p>code (Integer) – Refer to your Windows API documentation for the CBTProc function for information on this parameter.</p> <p>wp (Integer) – A 32 bit field depending on the code field.</p> <p>lp (Integer) – A 32 bit field depending on the code field.</p> <p>nodef (Short) – Refer to <i>Use of the nodef parameter</i>. In addition, the nodef parameter for this type of hook is only applicable to certain code values.</p> <p>An in-depth discussion of CBT hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this documentation has access to the Windows software development kit or Developer's Network CD-ROM.</p> <p>The lp parameter is frequently a pointer to a structure (whether this is the case, and the type of structure, depends on the code). The Windows hook control is aware of the types of structures supported, and ensures that the structure and contents are copied into the current memory address space - an important issue under Windows NT/2000/XP where address pointers may not be valid when moved between processes.</p>
DelayedEvent	<p>[VB] Sub WinHook1_DelayedEvent(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_DelayedEventEvent)</p>

	<p>Handles WinHook1.DelayedEvent [C#] void WinHook1_DelayedEvent(object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_DelayedEventEvent e)</p> <p>The PostEvent property is used to trigger this event. It is typically used to delay some operation without going through the trouble of setting up a timer. The CBTProcEvent fields are as follows:</p> <p>Ivalue (Integer) – Contains the value set into the PostEvent property that triggered this event.</p> <p>You may perform any operation during this event.</p>
JournalPlayProc	<p>[VB] Sub WinHook1_JournalPlayProc(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_JournalPlayProcEvent) Handles WinHook1.JournalPlayProc [C#] void WinHook1_JournalPlayProc (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_JournalPlayProcEvent e)</p> <p>This event is triggered for messages detected when the Monitor property is set to '6 - JournalPlayback'. The JournalPlayProcEvent fields are as follows:</p> <p>code (Integer) – Refer to your Windows API documentation for the JournalPlaybackProc function for information on this parameter.</p> <p>msg (Integer) – The keyboard or mouse message to place in the system queue.</p> <p>paramL (Integer) – A parameter depending on the message.</p> <p>paramH (Integer) – A second parameter depending on the message.</p> <p>wnd (Integer) – The window handle.</p> <p>mtime (Integer) – A 32 bit time stamp for the message.</p> <p>delay (Integer) – A 32 bit delay in milliseconds until the message will be placed in the system queue. Zero for no delay (default).</p> <p>An in-depth discussion of Journal hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this event has access to the Windows software development kit or Microsoft Developer's Network CD-ROM.</p>
JournalRecordProc	<p>[VB] Sub WinHook1_JournalRecordProc(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_JournalRecordProcEvent) Handles WinHook1.JournalRecordProc [C#] void WinHook1_JournalRecordProc (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_JournalRecordProcEvent e)</p>

	<p>This event is triggered for messages detected when the Monitor property is set to '7 - JournalRecord'. The JournalRecordProcEvent fields are as follows:</p> <p>code (Integer) – Refer to your Windows API documentation for the JournalPlaybackProc function for information on this parameter.</p> <p>msg (Integer) – The keyboard or mouse message to place in the system queue.</p> <p>paramL (Integer) – A parameter depending on the message.</p> <p>paramH (Integer) – A second parameter depending on the message.</p> <p>wnd (Integer) – The window handle.</p> <p>mtime (Integer) – A 32 bit time stamp indicating when the message was received.</p> <p>nodef (Short) – Refer to the <i>Use of the Nodef Parameter</i> section.</p> <p>An in-depth discussion of Journal hooks is beyond the scope of this manual. It is assumed that anyone wishing to use this event has access to the Windows software development kit or Microsoft Developer’s Network CD-ROM.</p>
<p>MessageProc</p>	<pre>[VB] Sub WinHook1_MessageProc(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MessageProcEvent) Handles WinHook1.MessageProc [C#] void WinHook1_MessageProc (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MessageProcEvent e)</pre> <p>This event is triggered for messages detected when the Monitor property is set to '2 - MessageProc' or '3 - SysMessageProc'. The MessageProcEvent fields are as follows:</p> <p>code (Integer) – Refer to your Windows API documentation for the JournalPlaybackProc function for information on this parameter.</p> <p>src (Short) – 0 if the message is inside a dialog box or message box.2 if the message is inside a menu.</p> <p>wnd (Integer) – The window handle.</p> <p>msg (Integer) – The message number.</p> <p>wp (Integer) – The wParam parameter.</p> <p>lp (Integer) – The lParam parameter.</p>

	<p>nodef (Short) – Refer to the <i>Use of the nodef parameter</i> section.</p> <p>nodef is only valid when the Notify property is set to '0 - When Hooked'.</p>
<p>MouseProc</p>	<pre>[VB] Sub WinHook1_MouseProc(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MouseProcEvent) Handles WinHook1.MouseProc [C#] void WinHook1_MouseProc (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MouseProcEvent e)</pre> <p>This event is triggered for messages detected when the Monitor property is set to '1 - MouseProc'. The MouseProcEvent fields are as follows:</p> <p>wnd (Integer) – The window handle.</p> <p>msg (Integer) – The message number.</p> <p>X (Integer) – The x location of the cursor in screen coordinates.</p> <p>Y (Integer) – The y location of the cursor in screen coordinates.</p> <p>hitcode (Integer) – A hit test code identifying the type of screen object at the position indicated. Refer to your Windows API reference or the on-line help for a list of these codes.</p> <p>peek (Short) – A value when non zero indicates that this message has been detected during a PeekMessage function call in which messages are not being removed (this means you will probably get duplicates of this message).</p> <p>nodef (Short) – Refer to the <i>Use of the nodef parameter</i> section .</p> <p>nodef is only valid when the Notify property is set to '0 - When Hooked'.</p>
<p>MouseProcLL</p>	<pre>[VB] Sub WinHook1_MouseProcLL(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MouseProcLLEvent) Handles WinHook1.MouseProcLL [C#] void WinHook1_MouseProcLL (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_MouseProcLLEvent e)</pre> <p>This event is triggered for messages detected when the Monitor property is set to '10 - MouseProcLL'. The MouseProcLLEvent fields are as follows:</p> <p>msg (Integer) – The message number.</p> <p>X (Integer) – The x location of the cursor in screen coordinates.</p> <p>Y (Integer) – The y location of the cursor in screen coordinates.</p>

	<p>mousedata (Integer) – Contains information on the mouse wheel or x button.</p> <p>flags (Integer) – Bit field that specifies the event-injected flag. Currently, Bit 0 is set if this message was injected.</p> <p>Time (Integer) – Time stamp for this message.</p> <p>Extrainfo (Integer) – Specifies extra information associated with the message.</p>
ShellProc	<pre>[VB] Sub WinHook1_ShellProc(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_ShellProcEvent) Handles WinHook1.ShellProc [C#] void WinHook1_ShellProc (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_ShellProcEvent e)</pre> <p>This event is triggered for messages detected when the HookType property is set to '8 - Shell'. The ShellProcEvent fields are as follows:</p> <p>code (Integer) – Refer to your Windows API documentation for the ShellProc function for information on this parameter.</p> <p>wp (Integer) – The wParam parameter.</p> <p>lp (Integer) – The lParam parameter.</p> <ul style="list-style-type: none"> ▪ <p>nodef (Short) – Refer to the <i>Use of the nodef parameter</i> section.</p>
WndMessage	<pre>[VB] Sub WinHook1_WndMessage(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_WndMessageEvent) Handles WinHook1.WndMessage [C#] void WinHook1_WndMessage (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_WndMessageEvent e)</pre> <p>This event is triggered for messages detected when the HookType property is set to '0 - GetMessage' or '4 - WindowProc'. The WndMessageEvent fields are as follows:</p> <p>wnd (Integer) – The window handle.</p> <p>msg (Integer) – The message number.</p> <p>wp (Integer) – The wParam parameter.</p> <p>lp (Integer) – The lParam parameter.</p> <p>nodef (Short) – Refer to the <i>Use of the nodef parameter</i> section.</p> <p>nodef is only valid when the Notify property is set to '0 - When Hooked'.</p>

<p>WndMessageRet</p>	<pre>[VB] Sub WinHook1_WndMessageRet(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_WndMessageRetEvent) Handles WinHook1.WndMessageRet [C#] void WinHook1_WndMessageRet (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_WndMessageRetEvent e)</pre> <p>This event is triggered for messages detected when the HookType property is set to '9 - WindowProcRet'. The WndMessageRetEvent fields are as follows:</p> <p>wnd (Integer) – The window handle.</p> <p>msg (Integer) – The message number.</p> <p>wp (Integer) – The wParam parameter.</p> <p>lp (Integer) – The lParam parameter.</p> <p>retval (Integer) – The return value parameter.</p> <p>nodef (Short) – Refer to the <i>Use of the nodef parameter</i> section.</p> <p>nodef is only valid when the Notify property is set to '0 - When Hooked'.</p>
<p>ForegroundIdleProc</p>	<pre>[VB] Sub WinHook1_ForegroundIdleProc(ByVal sender As Object, ByVal e As AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_ForegroundIdleProcEvent) Handles WinHook1.ForegroundIdleProc [C#] void WinHook1_ForegroundIdleProc (object sender, AxDesaware.SpyWorks.dwshkNET._DDwshkEvents_ForegroundIdleProcEvent e)</pre> <p>This event is triggered for messages detected when the HookType property is set to '11 - ForegroundIdleProc '. The ForegroundIdleProcEvent fields are as follows:</p> <p>nodef (Short) – Refer to the <i>Use of the nodef parameter</i> section.</p> <p>nodef is only valid when the Notify property is set to '0 - When Hooked'.</p>

.NET samples

SpyWorks samples are provided to demonstrate different functionality of SpyWorks technology. But, they are also provided for educational purposes. We hope that you'll be able to learn and benefit from the introduction of many advanced coding techniques found in our samples. In our samples, we attempt to write our code in compliance with Microsoft recommended coding practices. But, .NET is just beginning to evolve and this

may be a moving target as new coding practices may be introduced. We would like to have your feedback regarding which language you are using and which language you would prefer to have the sample code written in. We are still in the early stages of migrating to and learning .NET. If there are particular samples using some of the SpyWorks components or functions that you would like to see, please submit a request to support@desaware.com. Please include a detail description along with your contact information.

Differences between C# and Visual Basic .NET sample projects

In most cases, the Visual Basic .NET project was written first, then the C# project was written based on the Visual Basic .NET project. Visual Basic .NET specific functions are not used in favor of .NET equivalent namespaces when possible so that the code base between the two languages will be as similar as possible, making it easier to read the other language.

- BrowseFolder
 - Demonstrates how to use the shell's Browse Folder. As far as we can tell, there is no equivalent Browse Folder control or object in .NET.
- Clipboard
 - Demonstrates how to use .NET native subclassing to subclass the clipboard to detect when new data is available on the clipboard.
- ControlEsc
 - Demonstrates how to use SpyWorks Keyboard Hook to disable Control+Esc, Alt+Tab, and other system keys. Also demonstrates how to detect (but not disable) the Control+Alt+Del key.
- DeskTop
 - Demonstrates how to use SpyWorks Subclassing to subclass the System Desktop. Also demonstrates the use of the SpyWorks cross process functions.
- DetectNewWindows
 - Demonstrates how to use SpyWorks Windows Hook to detect newly created Windows for the entire system.
- EnumWin

- Demonstrates how to use .NET delegates as callback functions to enumerate all top level Windows on the system.
- ForeGroundIdle
 - Demonstrates how to use the ForeGroundIdle hook type to detect when your foreground thread is idle.
- Function Export
 - VB6 Export Functions
 - Demonstrates how to call Visual Basic 6.0 export functions from .NET. Demonstrates how to marshal parameters and structures passed to the functions. This sample is also helpful for calling other Windows API functions as it demonstrates how to Marshall parameters and using Platform Invoke.
 - .NET Export Functions
 - Demonstrates how to call .NET export functions from .NET and Visual Basic 6.0 applications. Demonstrates how to marshal parameters and structures passed to the functions and how to return Visual Basic 6.0 String data types (if your target caller is Visual Basic 6.0).
- KeyHook
 - Demonstrates how to use the SpyWorks Windows Hook to detect hot keys.
- Monitor
 - Demonstrates how to use the SpyWorks Windows Hook to detect keyboard and mouse activity for the entire system.
- MousePt
 - Demonstrates how to use the SpyWorks Windows Hook to track mouse movement to identify and display information on the Window the mouse is over.
- NativeSubclassing

- Demonstrates how to use .NET native subclassing to subclass the combo box control to detect when the combo has closed (CloseUp), and to subclass a text box to disable the default context menu.
- SetForeground
 - Demonstrates how to use the SpyWorks functions to force a Window to the foreground even if the process that Window belongs to is not the active process.
- ShellHook
 - Demonstrates how to use the Shell hook type to detect a variety of events related to Windows.
- SpyWin
 - Demonstrates how to enumerate all the Windows in the System. Organizes Windows by Process, Threads, and Parent Windows. Demonstrates how to retrieve additional information on a specified Window or search for an existing Window.
- TitleBar
 - Demonstrates how to use .NET native subclassing and how to call Windows API functions from .NET to custom draw a titlebar for your form.
- XTaskEditGetLine
 - Demonstrates how to use the SpyWorks functions to retrieve a line of text from a multi-line edit control located in another process.
- XTaskRichTextGet
 - Demonstrates how to use the SpyWorks functions to retrieve text from a rich text control located in another process.
- XTaskSubclass
 - Demonstrates how to use Windows API functions to add a new menu item to another application and SpyWorks Subclassing to detect when that menu item has been selected.