

The Desaware NT Service Toolkit COM Edition

Version 2.0

Windows NT 4, 2000, XP and Visual Basic 6

by

Desaware, Inc.

Rev 2.0.1 (06/05)

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 © 2000-2005 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 for use on a single computer. If you wish to transfer the license from one computer to another, you must uninstall it from one computer before installing it on the next. 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 or .dll) 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 © 2000-2005 Desaware, Inc. All rights reserved."

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

Files: You may include with your program a copy of the files dwNTServ.tlb, dwsvclnt.dll, dwsock6.dll, dwspyv6.dll, sockintf.dll, dwbkthrd.dll, cpapplet.tlb, dwscm.dll and dwspy5.dll. You may also distribute EXE and DLL files created using the Desaware service configuration wizard program and Desaware control applet configuration wizard program. You may **not** modify the files listed above in any way.

Source Files: Source code for portions of the Desaware NT Service Toolkit 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 toolkit 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 NT Service toolkit package.

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

Microsoft is a registered trademark of Microsoft Corporation. Visual Basic, Windows, Windows XP, Windows 2000 and Windows NT are trademarks of Microsoft Corporation. Desaware NT Service Toolkit, SpyWorks, StateCoder, VersionStamper, StorageTools, ActiveX Gallimaufry, Event Log Toolkit, Custom Control Factory, and SpyNotes #2, The Common Dialog Toolkit are trademarks of Desaware, Inc.

Limited Warranty

Desaware, Inc. warrants the physical medium (either diskettes or 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 diskette(s), CD 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 medium and documentation only (not the software) and is otherwise expressly and specifically disclaimed.

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

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

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

Table of Contents

TABLE OF CONTENTS	5
BEFORE YOU BEGIN	13
INTRODUCTION	14
NEW FEATURES FOR VERSION 2.0	15
<i>New Service Executable Command Line Options</i>	15
<i>New IdwServiceControl Methods and Properties</i>	15
<i>Improved Instrumentation and Diagnostics</i>	15
<i>New Features for Interactive Services</i>	15
<i>Improved Error Handling</i>	16
<i>Service Control Features</i>	16
<i>Other Features</i>	16
<i>Migration Support for .NET</i>	16
UPGRADING A VERSION 1.X SERVICE	16
WHAT IS AN NT SERVICE?	17
WHY A SERVICE?	17
TYPES OF SERVICES	17
<i>System Monitors</i>	17
<i>Background Tasks</i>	18
<i>Software Agent</i>	18
<i>Resource Pool</i>	19
<i>Business Objects</i>	19
HOW SERVICES DIFFER FROM REGULAR EXECUTABLES	19
SERVICES AND VISUAL BASIC 6	20
THE DESAWARE NT SERVICE TOOLKIT	21
LEARNING MORE	22
CREATING A SIMPLE SERVICE	24
STEP 1 – CONFIGURE THE SERVICE EXECUTABLE	24
<i>Service Executable Name</i>	24
<i>Service Component Name</i>	24
<i>Version Information</i>	24
<i>Thread Pool Size</i>	24
<i>Create VBR File</i>	25
STEP 2 – CREATE THE ACTIVE X DLL	25
STEP 3 – ADD THE SERVICE CONFIGURATION CLASS	25
STEP 4 – ADD THE SERVICE CLASS	27
STEP 5 – TEST AND RUN THE SERVICE	28

THE SERVICE FRAMEWORK MODEL.....	30
CONFIGURING THE SERVICE	32
IDWEASYSERVCONFIG METHODS	32
<i>AutoStart() As Boolean</i>	33
<i>ControlsAccepted() As enumServiceControls</i>	34
<i>DefaultTimes</i>	35
<i>GetDescription() As String</i>	35
<i>GetVersion</i>	36
<i>IgnoreStartupErrors</i>	36
<i>InteractWithDesktop() As Boolean</i>	36
<i>ServiceAccount() As String</i>	37
<i>ServiceAccountPassword() As String</i>	38
<i>ServiceDependencies() As String</i>	38
<i>ServiceProcessId() As Long</i>	39
IMPLEMENTING THE SERVICE CLASS	40
IDWEASYSERVICE METHODS RELATING TO STATE TRANSITIONS	41
<i>OnContinue</i>	42
<i>OnPause</i>	42
<i>OnStart</i>	42
<i>OnStop</i>	43
<i>OnShutdown</i>	43
IDWEASYSERVICE METHODS RELATING TO OTHER SERVICE CONTROL MANAGER EVENTS	45
<i>OnUserControlCode</i>	45
<i>OnParamChange</i>	45
<i>OnHardwareProfileChange</i>	46
<i>OnDeviceEvent</i>	46
<i>OnPowerRequest</i>	47
IDWEASYSERVICE METHODS SPECIFIC TO THE SERVICE FRAMEWORK	47
<i>OnTimer</i>	47
<i>WaitComplete</i>	48
IDWEASYSERVICE2 INTERFACE METHODS	48
<i>OnLogout</i>	48
IDWSERVICECTL - THE SERVICE CONTROL OBJECT.....	49
IDWSERVICECTL PROPERTIES	49
<i>InstallParameters (String)</i>	49
<i>StartupParameters (String)</i>	49
<i>Timeout (Long)</i>	49
<i>ControlsAccepted (Long)</i>	50
IDWSERVICECTL METHODS	50

<i>UpdateTransitionTime</i>	50
<i>StopService</i>	51
<i>SetWaitOperation</i>	51
<i>ClientExecuteBackground</i>	51
<i>ClearWaitOperation</i>	51
<i>GetInteractiveUser</i>	52
<i>RegisterApplicationObject</i>	52
<i>RegisterClientObjectName</i>	52
<i>RegisterDeviceNotification</i>	53
<i>UnregisterDeviceNotification</i>	53
<i>ReportEvent</i>	54
<i>ReportEvent2</i>	54
<i>Trace</i>	55
USING THE SERVICE CONFIGURATION PROGRAM	56
<i>Service Executable Name</i>	56
<i>Project Name</i>	57
<i>Version Information</i>	58
<i>Thread Count</i>	59
<i>Create VBR File</i>	59
<i>Compile Executable</i>	59
<i>Compile Completed</i>	59
RUNNING THE SERVICE CONFIGURATION WIZARD IN BATCH MODE	60
<i>Command Switches</i>	60
USING THE SERVICE EXECUTABLE LAUNCHER PROGRAM.....	63
BACKGROUND THREADS AND SYNCHRONIZATION OBJECTS	64
METHODS USED TO IMPLEMENT BACKGROUND THREADS	65
<i>Control Object (IdwServiceCtl Interface)</i>	65
<i>Service Object (IdwEasyService interface)</i>	67
EXPOSING SERVICE OBJECTS THROUGH COM AND DCOM	69
THE SERVICE FRAMEWORK OBJECT ARCHITECTURE.....	69
<i>Objects That Run on the Primary Service Thread</i>	69
<i>Objects That Run Within a Thread Pool Provided By the Service</i>	70
THE RUNNINGSERVICE OBJECT	70
CREATING THE APPLICATION OBJECT.....	72
<i>Reference Counting and the Application Object</i>	72
<i>Accessing the Service Object from the Application Object</i>	73
CREATING THE CLIENT OBJECT	74

THE IDWSERVICECLIENT INTERFACE AND THE UNUSUAL LIFE OF CLIENT OBJECTS	75
<i>Reference Counting and the Client Object</i>	75
<i>OnConnect</i>	80
<i>OnDisconnect</i>	80
<i>OnStop</i>	80
<i>ExecuteBackground()</i>	81
ADDITIONAL APPLICATION AND CLIENT OBJECT ISSUES	81
<i>Global Variables</i>	81
<i>Service State and the Client and Application Objects</i>	82
<i>Use OLE Callbacks, not Events</i>	82
SECURITY AND IMPERSONATION.....	83
NT/2000/XP SECURITY IN 250 WORDS OR LESS.....	83
IMPERSONATION	83
<i>Types of Impersonation</i>	84
CONFIGURING YOUR SERVICE FOR COM/DCOM CLIENT ACCESS	85
<i>Configuring the Service Access Through DcomCnfg</i>	86
<i>Configuring the Client System for Access to COM/DCOM Objects Exposed from the Service</i>	89
<i>Acting on Behalf of Clients</i>	91
THE DWSECURITY OBJECT.....	92
<i>Impersonate()</i>	92
<i>RevertToSelf()</i>	92
<i>GetUserInfo</i>	93
CREATING BACKGROUND THREADS.....	94
DWBACKTHREAD - QUICK START	94
DWBACKTHREAD – METHODS AND PROPERTIES	96
<i>LaunchObject(ObjectName As String) As Object</i>	96
<i>BackgroundExecute()</i>	96
<i>BackgroundExecuteDelayed()</i>	96
<i>BackgroundObject</i>	96
DWBACKTHREAD – SUMMARY OF RULES	97
EXAMPLES	99
COMMON ERRORS	102
INSTALLATION AND REGISTRATION.....	102
<i>Service Cannot be Deleted Error When Trying to Install or Delete a Service</i>	102
<i>Unable to Load Service Configuration Object Error When Trying to Install a Service</i>	102
DEBUGGING WITH VISUAL BASIC	102
<i>Permission Denied Runtime Error While Accessing the Service Control Object</i>	102

<i>Service Stops Suddenly Instead of a Runtime Error Appearing in your VB Code While Debugging</i>	102
CLIENT OBJECTS.....	103
<i>Permission Denied Error When Creating the RunningService Object</i>	103
<i>My Client Object is not Receiving an OnStop Method Call</i>	103
<i>A Method Works in VB Debug Mode, but Fails When Compiled</i>	103
<i>Unable to Access an Object Through DCOM</i>	103
WHILE RUNNING	104
<i>The Service Stops Working</i>	104
<i>The Service Cannot Access Network Resources When Running as a Service</i>	104
LICENSING ISSUES	105
TESTING AND DEBUGGING	106
TRACING AND LOGGING	106
TESTING AND DEBUGGING – SIMULATOR MODE	107
TESTING AND DEBUGGING – WHILE RUNNING AS A SERVICE.....	108
TESTING AND DEBUGGING – FULL NATIVE MODE	108
TESTING COM AND DCOM	110
THE DWSCM COMPONENT: SERVICE CONTROL MANAGER.....	111
DWSCM ARCHITECTURE.....	111
DWSERVICEMANAGER METHODS.....	112
<i>InitializeSCManager</i>	112
<i>EnumServicesStatus</i>	112
<i>OpenService</i>	113
<i>GetDisplayNameFromServiceName</i>	113
<i>GetServiceNameFromDisplayName</i>	113
<i>CreateService</i>	114
<i>LockServiceDatabase () As Boolean</i>	114
<i>UnlockLockServiceDatabase () As Boolean</i>	114
<i>QueryLockStatus</i>	115
DWSERVICEOBJECT METHODS AND PROPERTIES.....	115
<i>StartService</i>	115
<i>ControlService</i>	116
<i>QueryServiceStatus() As dwServiceStatus</i>	116
<i>QueryServiceConfig() As dwServiceConfig</i>	116
<i>ChangeServiceConfig</i>	117
<i>EnumDependentServices</i>	117
<i>DeleteService()</i>	117
<i>ServiceName as String</i>	118
<i>ServiceHandle as Long</i>	118

DWSERVICESTATUS PROPERTIES	118
<i>DisplayName as String</i>	118
<i>ServiceName as String</i>	118
<i>CurrentState as ServiceStateConstants</i>	118
<i>ControlsAccepted as ControlsAcceptedFlags</i>	118
<i>Win32ExitCode as Long</i>	118
<i>ServiceSpecificExitCode as Long</i>	118
<i>CheckPoint as Long</i>	118
<i>WaitHint as Long</i>	119
DWSERVICECONFIG PROPERTIES	119
<i>ServiceType as ServiceTypes</i>	119
<i>StartType as ServiceStartTypes</i>	119
<i>ErrorControl as ServiceErrorControlType</i>	119
<i>BinaryPathName as String</i>	119
<i>LoadOrderGroup as String</i>	120
<i>TagId As Long</i>	120
<i>Dependencies as String</i>	120
<i>AccountName as String</i>	120
<i>Password as String</i>	120
<i>DisplayName as String</i>	120
<i>Description as String</i>	121
ENUMERATIONS AND CONSTANTS	121
<i>ServiceTypes Enumeration</i>	121
<i>ServiceStartTypes Enumeration</i>	121
<i>ServiceErrorControlTypes Enumeration</i>	122
<i>ServiceControlRights Enumeration</i>	122
<i>ServiceAccessRights Enumeration</i>	123
<i>ServiceControlConstants Enumeration</i>	124
<i>ServiceStateConstants Enumerations</i>	125
<i>EnumServiceStates Enumeration</i>	125
<i>ControlsAcceptedFlags Enumeration</i>	125
THE DWSOCK COMPONENT: WINSOCK PROGRAMMING	127
HOW TO APPROACH THE WINSOCK PACKAGE	127
IMPORTANT NOTE REGARDING SUPPORT FOR THIS COMPONENT	128
LEARNING WINSOCK	128
<i>IP Addressing</i>	129
<i>Ports</i>	129
<i>UDP and TCP</i>	130
DWSOCK ARCHITECTURE	130
<i>Dependencies</i>	132
USING THE WINSOCK PACKAGE	132

WINSOCK UTILITY FUNCTIONS.....	133
<i>Obtaining Winsock Version Information</i>	133
<i>Obtaining Your Host Name</i>	133
<i>Determine the Standard Port Number of a Service</i>	133
<i>Perform an Asynchronous Name Resolution</i>	133
HTTP EXAMPLE.....	134
CREATING CONTROL PANEL APPLETS	136
BUILDING A CONTROL PANEL APPLETT	136
USING THE CONTROL PANEL APPLETT WIZARD PROGRAM.....	136
<i>Control Panel Applet Name</i>	136
<i>Project Name</i>	137
<i>Version Information</i>	138
<i>Description</i>	138
<i>Icon File</i>	139
<i>Compile Applet</i>	139
<i>Compile Completed</i>	140
<i>System Compatibility</i>	140
CREATE AN ACTIVE X DLL FOR YOUR CONTROL PANEL APPLETT	140
<i>CplDbtClk</i>	140
<i>CplExit()</i>	141
<i>CplGetCount() As Long</i>	141
<i>CplInit() As Long</i>	141
<i>CplInquire</i>	141
<i>CplNewInquire</i>	142
<i>CplStartWParms</i>	143
<i>CplStop</i>	143
USING CONTROL PANEL APPLETS WITH SERVICES	143
INSTALLING AND TESTING YOUR CONTROL PANEL APPLETT	144
<i>Installing the CPL File on Windows 2000/XP</i>	145
DISTRIBUTING YOUR CONTROL PANEL APPLETT	146
INSTALLING AND DISTRIBUTING YOUR SERVICE	147
COMPILING YOUR COMPONENT.....	147
CONFIGURING SECURITY	147
CONFIGURING REMOTE SYSTEMS TO ACCESS OBJECTS FROM YOUR SERVICE.....	148
SERVICE EXECUTABLE COMMAND LINE OPTIONS	148
REDISTRIBUTABLE COMPONENTS	149
VERSION VERIFICATION	150
TECHNICAL SUPPORT.....	151
FRAMEWORK RESTRICTIONS.....	152

CONFIGURATION ISSUES	152
DIFFERENCES BETWEEN THE LIGHT EDITION AND STANDARD EDITION OF THE DESAWARE NT SERVICE TOOLKIT	153
OTHER SOURCES OF INFORMATION	154
<i>www.desaware.com</i>	154
<i>Dan Appleman's Visual Basic Programmer's Guide To The Win32 API</i>	154
<i>Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed</i>	154
<i>Dan Appleman's Win32 API Puzzle Book and Tutorial for Visual Basic Programmers</i> ...	155
<i>Windows API Online Help</i>	155
<i>Microsoft's Developers Network CD Rom</i>	156
<i>Microsoft's Windows Software Development Kit and Win32 Software Development Kit</i> ..	156
INDEX	157
DESAWARE PRODUCT DESCRIPTIONS	162

Before You Begin

The Desaware NT Service Toolkit is available in four editions.

- | | |
|---|--|
| Full (COM) Version | The full edition includes all of the features described in this documentation and allows the creation of services using Visual Basic 6. |
| Light Version | Included with SpyWorks Professional 6.2 or later. Replaces the previous service class library. Does not support client or application objects. Does not support automatic background threads for waiting on NT synchronization objects. It does not support version 2.0 features. |
| Demo Version (available for both COM and .NET editions) | <p>Includes all the features described in this documentation except for the service configuration program, the service executable launcher program, and the control panel applet wizard program. This means you can experiment with all of the sample services and develop your own – as long as they use the EasyServ.exe service executable we provide, and use a VB component class named dwEasyServ (thus you can only experiment with one service at a time). You cannot create or distribute your own services with this version.</p> <p>The demo version has an expiration date. Once it expires you can download a new version from Desaware's web site.</p> |
| .NET Version | This is a separate product which provides all of the capabilities of the COM edition (in addition to several new features). This edition is designed for use with Visual Basic .NET, C# or other .NET languages, and does not allow creation of services using VB6. |

Introduction

The Desaware NT Service Toolkit is designed to make it easy to create reliable and supportable NT services using Visual Basic. For those familiar with previous NT service implementations offered as part of Desaware's SpyWorks, you will find this new technology to be both more reliable and infinitely easier to use. In fact, you'll be able to create solid services in less than 5 minutes.

The Desaware NT Service Toolkit offers the following features:

- True NT service allows detection and response to all service handler requests.
- Ability to configure all service parameters.
- Supports latest Windows 2000/XP features, but remains compatible with NT 4.
- Ability to use COM components including database and Winsock from within your service.
- Simulator allows testing and debugging of service as a standalone executable before deployment as a service.
- Full debug and trace support of the service within the Visual Basic IDE while it is running as a service.
- Easy to use background thread waiting on any NT synchronization object.
- Familiar event driven architecture with built in timer support.
- Service controlled state transition timeouts (start, stop, pause, continue).
- Ability to control a running service using COM or DCOM.
- Expose COM objects on a thread pool for high scalability.
- Support for impersonation of clients (act on behalf of clients).
- Launch arbitrary background threads for asynchronous operations.
- Perform asynchronous operations on client threads.
- Create control panel applets using Visual Basic for configuring your service.

New Features for Version 2.0

New Service Executable Command Line Options

-User and **-Password** allow you to specify the account in which the service will run. This setting overrides that provided in the Service Configuration file, and is ideal for cases where the account must vary from system to system. These options are not supported for services that are set to interact with the desktop.

-Params allows you to set parameters during installation which can be read at any time by the service when it runs.

-Silent prevents any message boxes from being displayed during installation operations, improving support for remote and automated installs.

New IdwServiceControl Methods and Properties

StartupParameters – Allows you to read parameters set during manual startup of a service via the control panel or Service Control Manager.

InstallParameters – Allows you to read parameters specified in the command line when the service is installed.

Trace – Allows you output arbitrary text from your service component to the framework tracing system for diagnostic purposes (please refer to the following section).

GetInteractiveUser – Obtains the account information for the user currently logged on.

Improved Instrumentation and Diagnostics

Framework logging – Definable trace levels control logging of detailed information about the operation of the framework to help resolve configuration issues.

New Features for Interactive Services

New IdwEasyService2 interface provides an OnLogout method that allows you to determine when an interactive user has logged off the system.

New features allows you to determine (in most cases) if an interactive user is logged on and to retrieve their account name and domain.

Improved Error Handling

Robust trapping and detection of runtime errors that occur in your Visual Basic component allow a cleaner shutdown of services when errors occur. Improved diagnostics allow reporting of where and when errors occur, making debugging of services much easier.

Service Control Features

New VB6 classes demonstrate how to control services (including starting, stopping and sending information to running services). Full source code for these classes is included.

Other Features

Improved and earlier detection and handling of system shutdown.

Improved default security handling reduces the amount of configuration needed in remote and DCOM based scenarios.

Migration Support for .NET

Services created with the COM edition of the NT Service Toolkit can be migrated easily to the .NET edition of the NT Service Toolkit. The .NET edition uses the same overall architecture including virtually identical interfaces.

Simply migrate your existing VB6 classes to VB.Net or C#, and they will plug in directly to the .NET framework. Plus, the .NET version of your services will automatically continue to support access by today's COM and DCOM clients as well as access from .NET clients using .NET remoting.

Upgrading a Version 1.x Service

You can upgrade your  service framework to the 2.0 service framework using the following steps:

- Uninstall your service.
- Open your service project. Add code for any new service properties, methods or events that you desire.
- Compile your service project.
- Create your service executable.
- Install your service.

What is an NT Service?

An NT service is a regular Windows executable. In most ways it works the same way as a standard application. But there are a few important differences both in the way they work and how they are built internally.

Why A Service?

NT services are intended to be, in a sense, a part of the operating system. An operating system provides services to applications. Services define additional areas of functionality that extend the capability of the operating system. For example: the NT event log, login program, telephony and web server are all services. Services can be automatically started by the operating system when a system is started. Developers create NT services for many different reasons. Ultimately, a service has only three real advantages over a regular executable:

- A service can be configured to start up automatically on system boot, and can have its operation controlled by the system (either locally or remotely).
- A service can run without a user logging on, and can continue running as users log in and off a system.
- A service can run under the security context of your choice, allowing it to perform operations independently of who is logged on, or which client is accessing it.

Yet these advantages allow a wide variety of operations that are best performed in services.

Types of Services

NT services tend to fall into certain categories. Keep in mind that a single service might actually fall into multiple categories.

System Monitors

As a multi-tasking system, there are many things going on at any given time while Windows is running. This is especially true on servers, where client systems might be modifying files, the registry, or performing a wide array of other tasks.

Windows supports a variety of synchronization objects that can be used to monitor these tasks. These objects can be used to detect changes to files or directories, for changes to the registry, for the termination of running applications, and many more events.

Unfortunately, it is very inefficient to monitor these objects using Visual Basic, because VB requires that you poll the object continuously to watch for changes - a very inefficient approach.

The Desaware NT Service Toolkit monitors any objects you choose using a background thread that is completely suspended while you are waiting for an event to occur. As a result, your service can enter a highly efficient wait state, while still being able to respond to client or service control requests.

Background Tasks

Consider a business object that runs on a server. If you implement it as a multi-use ActiveX EXE, the server will launch the EXE on receipt of a client request, and free it once all clients are finished with the object.

But what if the business object has a long initialization process that it must go through before it can respond to a client request? This is not at all uncommon on corporate systems. To go through the initialization each time a request is received is prohibitive.

A service can solve this problem in two ways. First, since a service can be configured to launch automatically when a system is started, you can implement your business objects in the service and simply perform the initialization at a more convenient time, when the system boots. An alternate approach is to have the service launch your EXE server and hold a reference to it (keeping it open), then monitor the server. If the EXE server is terminated (due to a crash or error), the service can detect that condition and automatically restart the EXE server.

Services are also ideal for scheduled operations that run in the background. Built-in waitable timer support allows services you create to easily perform operations on a scheduled or periodic basis.

Software Agent

The Desaware NT Service Toolkit allows you to expose client objects from your service that are accessible through COM or DCOM. These objects can work in two ways.

Normally, they run under the same account as the service itself (you can, of course, decide the account under which the service runs). Let's say you have a critical operation that you don't want to allow users to access directly. Since the client objects run in the service account, you can have it act as an agent for the user - performing operations that the user is not allowed to do.

At other times, however, you might want the client object to perform operations as if it were logged in to the user's account. Perhaps to allow it to access information that is personal to the user, or perhaps to prevent access to resources that the user does not have permission to touch. The Desaware NT Service Toolkit allows your client objects to "impersonate" the user in these situations. You can turn impersonation on and off on a line by line basis.

Resource Pool

Another common use for services is to make it easy for clients to share information, or to control access to a limited set of resources.

The Desaware NT Service Toolkit allows you to define an application object that is global to all clients using the service (even though the client objects may be running in separate threads). This makes it easy for clients to share information, for the service to hold information for clients, or for clients to communicate with each other.

Business Objects

The Desaware NT Service Toolkit takes advantage of Visual Basic's strength in creating components - in fact, all you need to do to create your service is to create a new ActiveX DLL server and add a few predefined classes. You'll find it easy to incorporate your existing business objects into services, or to call them from your service.

How Services Differ from Regular Executables

When a regular application is launched, it is initially assigned a single thread. The application can create additional threads, but these threads remain entirely under the control of the primary application.

A service application uses at least two threads. The primary thread of the service belongs to the service executable (and it can create additional threads). In addition, a handler thread is used by the operating system to control the service and notify it of system requests including instructions to pause, stop or resume operation.

Services and Visual Basic 6

The dual thread architecture imposes a specific set of requirements on application programmers using C++ in terms of the design of the service and synchronization between the primary service thread and the handler thread. Unfortunately it imposes a more severe set of restrictions on Visual Basic programmers because Visual Basic applications are required to adhere to the OLE threading rules for the STA (single threaded model) COM objects. In other words: Visual Basic is not thread safe.

This leaves Visual Basic programmers with three options:

- Use a wrapper program that generically allows any executable to run as a service.
- Create a class architecture that violates the OLE rules in a limited and well documented manner.
- Use Visual Basic .NET or C#.

The first of these approaches suffers from serious limitations. The executable is not a true NT service, and typically cannot respond well to notifications from the handler thread. This approach thus limits you to just starting and stopping the service using default timeouts, and uses a standard set of options instead of allowing you to take advantage of all of the configuration options available to services. Services created in this manner are also difficult to debug.

The second of these approaches is used by the SpyWorks NT service class library (that is superceded by this toolkit). It offers the full flexibility of a true NT service, but because it pushes the limits of Visual Basic's threading support, it does not offer full capability to access generic COM objects from the service.

The third of these approaches, using Visual Basic .NET or C#, is quite reasonable for those who are ready to deploy the .NET framework and have already learned .NET (which does have a substantial learning curve).

While Visual Studio .NET does include a basic framework for creating services, it lacks many of the features of this toolkit. A .NET edition of the Desaware NT Service Toolkit is available that provides VB .NET and C# developers with all of the features of this framework (and then some), in an architecture that is highly compatible with the COM edition. This allows developers to migrate their VB6 based services quickly to VB .NET, or to develop their services now with the familiar VB6 platform and migrate later when they are ready to move to .NET. Please refer to Desaware's web site for a complete list of features for the .NET edition of this toolkit.

The Desaware NT Service Toolkit

This toolkit overcomes the limitations of the previously discussed approaches by defining a framework that appears to the operating system as a 100% standard NT service, and simultaneously appears to a Visual Basic component as a standard COM client application.

The approach follows from the following chain of reasoning:

- Visual Basic is a terrible tool for creating a service executable because of threading issues.
- Visual Basic is an outstanding tool for creating COM components.
- A service executable can both use and expose COM components.
- COM components used by a service can provide part of the functionality of a service.

What would happen then, if you increased the proportion of service functionality provided by the COM object to the point where virtually all of the service was, in fact, implemented by that object?

You'd have the Desaware NT Service Toolkit.

An executable file is configured using a utility that we provide. This service executable contains minimal information about the service. It supports the main thread, handler thread, additional threads used to wait on NT synchronization objects, and additional threads to support client objects.

Your service functionality is defined in a Visual Basic component that you create which is loaded by the service executable. Your component contains at least two objects, one that is used to configure the service, the other that implements the service itself. These objects implement two standard interfaces that are defined by the toolkit, `IdwServiceConfig` for configuration, and `IdwService` to control the service itself. Your component can be easily debugged using the Visual Basic development environment even while running as a service. In addition, you can expose an application object that is shared by all service clients, and additional client objects.

The service framework automatically handles context switches to ensure that calls to your service objects do not violate the OLE threading rules.

If this all sounds confusing, don't worry. The tutorial that follows will guide you through the process of building a service, and in the process help you become familiar with the framework and its features.

Learning More

The documentation provided here is intended to address the needs of most Visual Basic programmers who wish to write services using the framework. However, writing services touches on virtually every aspect of Windows programming from COM to DCOM to security to the service API itself. A document that covered everything you could possibly need to know would fill several volumes, and be nearly impossible to navigate. This manual will discuss the use of a variety of technologies in the context of services and the service framework, but will not discuss those technologies at length. Thus, for example, we'll show you how to expose an object so it can be accessed through DCOM, and mention some of the issues that relate to use of DCOM in services. But you'll have to look elsewhere for in-depth information on DCOM and how to configure DCOM for use in your enterprise.

At a bare minimum, we will assume that you already know the following:

- You know Visual Basic programming at least on an intermediate level.
- You know how to create ActiveX DLL's and configure their properties.
- You have a fundamental understanding of concepts including COM, threading, use of API functions, and process spaces.

We strongly encourage you to read at least the first two parts of Dan Appleman's book "Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed". It will help you to understand and take full advantages of the features of this toolkit. You can also find additional articles on related subjects such as multi-threading on Desaware's web site at www.desaware.com.

Creating A Simple Service

There are five steps in creating and testing a service using the Desaware NT Service Toolkit.

Step 1 – Configure the Service Executable

The Service Executable is created using the Desaware NT Service Configuration Wizard program. This wizard will prompt you for the following information:

Service Executable Name



is the name of the service executable file. It can be any name you choose.

Service Component Name

This is the name of the COM object that the service will launch. Use the project name of the Visual Basic DLL that you will create. If you decide later that you will want to change the project name, you will have to run this configuration program again.

You should choose a component name that is unique – duplication of project names used by this framework can cause services to fail to work properly. We recommend including your company name or initials in the name. For example: most Desaware components include the prefix “dw”.

Version Information

This is where you set the version information for the service executable. You can set most standard Windows version information fields.

Thread Pool Size

(Not Available in Light Edition)

If your service will expose COM objects for use by clients using the service, those objects will be created on a thread pool so that their operation will not interfere with the primary service. You can set the size of the thread pool with this option.

Create VBR File (Not Available in Light Edition)

If your service will expose objects that clients can access through DCOM, you'll need a VBR file which is used by the clireg32 application to create the necessary registry entries for accessing your service objects remotely. The configuration program will create a VBR file for you automatically upon request.

Step 2 – Create the ActiveX DLL

Create a new ActiveX DLL project that has the project name that you specified earlier when creating the service executable.

Using the Project References dialog, add a reference to the “Desaware NT Service Toolkit 1.0 Type Library” (dwNTServ.tlb).

Step 3 – Add the ServiceConfiguration Class

Create a new class and name it “ServiceConfiguration”.

Add the following code to the class (the easiest way is to add the ServiceConfiguration class, ServiceCfg.cls from the "Template" directory):

```
Private Declare Function GetCurrentProcessId Lib _
    "kernel32" () As Long
Implements IdwEasyServConfig

Private Function IdwEasyServConfig_AutoStart() As _
Boolean
End Function

Private Function _
    IdwEasyServConfig_ControlsAccepted()_
    As EASYSERVLib.enumServiceControls
    IdwEasyServConfig_ControlsAccepted = _
    svcShutdown Or svcStop
End Function

Private Sub _
    IdwEasyServConfig_DefaultTimes(DefaultStartTime As _
    Long, DefaultStopTime As Long, DefaultPauseTime As _
    Long, DefaultContinueTime As Long)
End Sub

Private Function IdwEasyServConfig_GetDescription() _
    As String
    IdwEasyServConfig_GetDescription = _
```

```

        "Place your service name here" & Chr$(0) & _
        "Place description here"
End Function

Private Sub IdwEasyServConfig_GetVersion( _
    MajorVersion As Long, MinorVersion As Long)
    MajorVersion = App.Major
    MinorVersion = App.Minor
End Sub

Private Function _
    IdwEasyServConfig_InteractWithDesktop() As Boolean
End Function

Private Function IdwEasyServConfig_ServiceAccount _
    () As String
End Function

Private Function _
    IdwEasyServConfig_ServiceAccountPassword() _
    As String
End Function

Private Function _
    IdwEasyServConfig_ServiceDependencies() As String
End Function

Private Function _
    IdwEasyServConfig_IgnoreStartupErrors() As Boolean
End Function

Private Function _
    IdwEasyServConfig_ServiceProcessId() As Long
    IdwEasyServConfig_ServiceProcessId = _
    GetCurrentProcessId()
End Function

```

The ServiceConfiguration class implements the IdwEasyServConfig interface which is used by the service to retrieve configuration information from your DLL. The majority of these functions can be left empty. The important ones are:

- | | |
|------------------|--|
| ControlsAccepted | Determines whether your service accepts Pause, Continue, Stop and other service commands. |
| GetVersion | This allows the service executable to obtain the current version number from your component. |

ServiceProcessId This is used by the service executable to determine whether your component is running in process or out of process. The only time your service component will run out of process is while debugging. This allows the service executable to perform various tasks to help make debugging easier.

Step 4 – Add the Service Class

Create a new class and name it “Service”. Add the following code to the class (the easiest way is to add the Service class, Service.cls from the “Template” directory):

```
Implements IdwEasyService

Private Function IdwEasyService_OnContinue( _
    ByVal ControlObject As EASYSERVLib.IdwServiceCtl) _
    As Long
End Function

Private Function IdwEasyService_OnPause(ByVal _
    ControlObject As EASYSERVLib.IdwServiceCtl) As Long
End Function

Private Function IdwEasyService_OnShutdown(ByVal _
    ControlObject As EASYSERVLib.IdwServiceCtl, _
    StopPending As Boolean) As Long
End Function

Private Function IdwEasyService_OnStart(ByVal _
    ControlObject As EASYSERVLib.IdwServiceCtl) As Long
End Function

Private Function IdwEasyService_OnStop(ByVal _
    ControlObject As EASYSERVLib.IdwServiceCtl) As Long
End Function

Private Sub IdwEasyService_OnTimer(ByVal _
    ControlObject As EASYSERVLib.IdwServiceCtl)
End Sub

Private Sub IdwEasyService_OnUserCode(ByVal _
    ControlObject As EASYSERVLib.IdwServiceCtl, _
    ByVal ControlCode As Integer)
End Sub

Private Sub IdwEasyService_WaitComplete(ByVal _
    ControlObject As EASYSERVLib.IdwServiceCtl, ByVal _
```

```

ThreadID As Long, ByVal CompletionType As Long, _
ByVal ObjectIndex As Long)
End Sub

Private Sub IdwEasyService_OnParamChange(ByVal _
ControlObject As EASYSERVLlib.IdwServiceCtl)
End Sub

Private Function _
IdwEasyService_OnHardwareProfileChange(ByVal _
ControlObject As EASYSERVLlib.IdwServiceCtl, _
ByVal ChangeType As Long, ByVal ChangedData As _
Long) As Boolean
End Function

Private Function IdwEasyService_OnDeviceEvent _
(ByVal ControlObject As EASYSERVLlib.IdwServiceCtl, _
ByVal EventType As Long, ByVal EventData _
As Long) As Boolean
End Function

Private Function IdwEasyService_OnPowerRequest _
(ByVal ControlObject As EASYSERVLlib.IdwServiceCtl, _
ByVal APMessage As Long, ByVal Flags As Long) _
As Boolean
End Function

```

Step 5 – Test and Run the Service

You'll typically want to test the service as a standalone executable before installing it as a service. This is because it is much easier to install and test standalone executables. To do so, just do the following:

- Run your service component in the Visual Basic environment. (If prompted, specify to wait until the components are created option.)
- Register the service executable by running the program a command line using the parameters “-RegServer”.
- Run the service executable again, this time using the parameter “-Sim”.

The service will begin to run – if you run our Beeper Service sample, you should hear periodic beeps as the built-in timer executes. The service simulator window allows you to exercise the service using standard service controls.

To install as a service, all you need to do is run the executable with the parameter “-i”.

Remember that you need to either have your component running in the VB environment, or compiled and registered, or you will see a service initialization error.

That's all it takes to create a basic service. Everything else builds on this basic framework.

The Service Framework Model

Figure 1 depicts an outline of the core section of the service framework (you'll see a more complete illustration of the framework later when you learn about the COM features of the framework).

The service executable uses your ServiceConfiguration object to obtain configuration information for the service. It uses your Service object to notify your service of key events in the lifetime of the service.

At the same time, your Service object has access to the framework's ServiceControl object which offers extended functionality to your service component.

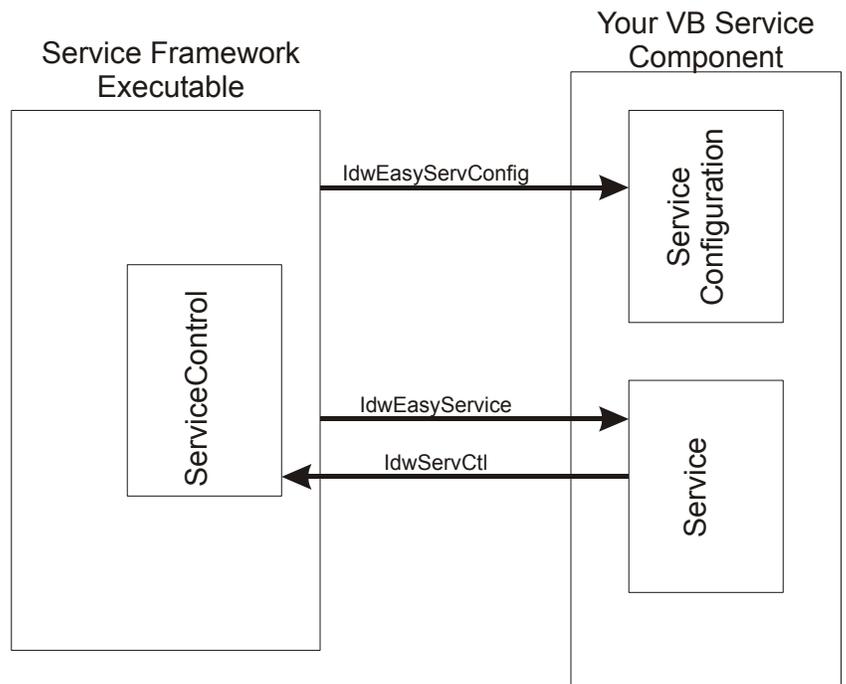


Figure 1
Core Section of the Service Framework

In each case, communication between the component and the service takes place on a private interface that is defined in the file `dwNTServ.tlb` which was registered when you installed the Toolkit (you will need to distribute this file with your service if you intend to expose objects from your service). Use of private interfaces ensures maximum performance. The next few sections cover each of these interfaces in-depth, and describe how each of these objects interacts with the others.

Configuring The Service

The ServiceConfiguration class is used to provide to the service executable the information that it needs to configure the service. The object is created by the service executable both during registration, and when the service is about to run.

NOTE: If you had successfully installed a service and you make any ServiceConfiguration changes, you must uninstall the service and then install it again in order for those changes to take effect.

Refer to the MSDN documentation for services for a more in-depth explanation of the various service configuration parameters.

The object should implement no methods other than those defined by the IdwEasyServConfig interface. The interface is implemented by the object by adding the following line to the class module:

```
Implements IdwEasyServConfig
```

Summary

Class name: ServiceConfiguration

Instancing: Multi-Use

Implements: IdwEasyServConfig

Rules: Do not add public methods or properties other than those required by the IdwEasyServConfig interface.

All methods of the interface must be implemented. Empty method declarations are sufficient except for the GetDescription, GetVersion and ControlsAccepted methods.

IdwEasyServConfig Methods

The IdwEasyServConfig interface includes the following methods. All must be implemented in the function, however only the GetDescription, GetVersion and ControlsAccepted methods require that you include any code in the function.

AutoStart() As Boolean

The service framework supports two options for service startup: Automatic startup after the system starts, and startup on demand (via the service control manager). These correspond to the SERVICE_AUTO_START and SERVICE_DEMAND_START options. There are three other startup options that are not supported by this framework. Two of them are usable only by system devices or drivers. The final, SERVICE_DISABLED, can be handled by manually disabling the service after installation (it was a consensus amongst our developers that installing a disabled service was rather pointless).

The default for the service framework is “start on demand”.

If you return True as the result of this method call, your service will be installed to start automatically when the system starts. To do so, simply add this line to the method:

```
IdwEasyServConfig_AutoStart = True
```

Important Notes: On a system using NT 4.0 Service Pack 5 or later, setting IdwEasyServConfig_AutoStart to True may cause your Service to fail on system start up. If the Event Log displays the following two error messages - “Timeout (120000 milliseconds) waiting for service to connect.” and “The service did not respond to the start or control request in a timely fashion.”, then you will need to add *ONE* of the following lines of code to the IdwEasyServConfig_ServiceDependencies function.

```
IdwEasyServConfig_ServiceDependencies = "Browser"
```

```
IdwEasyServConfig_ServiceDependencies =  
"ProtectedStorage"
```

```
IdwEasyServConfig_ServiceDependencies = "Replicator"
```

```
IdwEasyServConfig_ServiceDependencies = "RasMan"
```

or

```
IdwEasyServConfig_ServiceDependencies = "RasAuto"
```

ControlsAccepted() As enumServiceControls

Every service can be controlled to some degree by the system through the service control manager. The service control manager can be accessed via the control panel or system administration tools. You can decide which controls your service will accept from the service control manager.

The service framework includes an enumeration named `enumServiceControls` that defines the possible ways that the service control manager can interact with your service, and allows you to decide which controls to accept.

<code>svcStop = 1</code>	Service accepts Stop commands.
<code>svcPauseAndContinue = 2</code>	Service accepts Pause/Continue commands.
<code>svcShutdown = 4</code>	Service accepts Shutdown.
<code>svcParamChange = 8</code>	Service accepts parameter changes (Win2K only).
<code>svcHardwareProfile = 0x20</code>	Service accepts hardware profile changes (Win2K only).
<code>svcPowerEvent = 0x40</code>	Service accepts power events (Win2K only).

To allow your service to accept commands from the service control manager, return the allowed commands by using the Or operator to combine values from the `enumServiceControls` enumeration.

For example, to accept the Stop, Pause and Continue commands, you would add the following line to the `ControlsAccepted` method:

```
IdwEasyServConfig_ControlsAccepted = svcStop Or  
svcPauseAndContinue
```

We strongly recommend that every service you implement at least accept the `svcStop` command. Services that do not accept this command will have no way to perform internal cleanup, and will continue to run until you shut down your system.

You can change the commands accepted by the service while it is running using the `ControlsAccepted` property of the `ServiceControl` object provided by the service framework.

DefaultTimes (DefaultStartTime As Long, DefaultStopTime As Long, DefaultPauseTime As Long, DefaultContinueTime As Long)

When the service control module sends commands to the service, it allows a certain amount of time for the service to respond that it has completed the specified task before it assumes that an error occurred. The time can be specified by the service.

This method allows you to set the default timeout value for the Start, Stop, Pause and Continue commands in milliseconds. The service can extend the time using the `ServiceControl` object – this only sets the initial default value – the amount of time that the system will allow for you to respond to the various commands the first time they are called.

The default timeouts if you do not add code to this method is 15 seconds each. The minimum timeout value is 2 seconds. If you specify any time less than 2000, the number will internally be set to 2000.

GetDescription() As String

This method allows your service to specify its display name and description. These strings will appear in system utilities that control services. Choose any descriptive strings that are no longer than 255 characters.

NOTE: Although the function name would indicate that this function sets the Description for your service, this function actually sets the Display Name for your service. NT 4.0 only supports service Display Names, Windows 2000/XP supports both service Display Names and Descriptions.

Set the display name using the following code:

```
IdwEasyServConfig_GetDescription =  
"Place your display name here"
```

Set the display name and description by appending the description and using a null character to separate the two strings.

```
IdwEasyServConfig_GetDescription =  
"Place your display name here" & Chr$(0) & _  
"Place your description here"
```

GetVersion

(MajorVersion As Long, MinorVersion As Long)

This method allows the service framework to obtain the version number of the service from the component. Add the following code to this method:

```
MajorVersion = App.Major  
MinorVersion = App.Minor
```

These values are displayed when you use the “-v” extension to display the version of an installed version. Note that Visual Basic does not automatically increment the Minor version number – so you may want to form the MinorVersion number as a combination of the Visual Basic minor and revision numbers.

IgnoreStartupErrors

A service can specify how the system should react when the service does not start correctly. The service framework supports two options: the normal response is for the system to log the error and display a warning message box indicating that a service failed to load. If you return True as a result to this method, the system will still log the error, but will not display a message box. To ignore errors, use the following code:

```
IdwEasyServConfig_IgnoreStartupErrors = True
```

InteractWithDesktop() As Boolean

This method allows you to indicate that this service is able to interact with the desktop of the user who is currently logged in. The default is that the service may not interact with the desktop. To enable interaction, use the following code:

```
IdwEasyServConfig_InteractWithDesktop = True
```

Refer to the MSDN documentation for more information on use of interactive services. Additional limitations apply to interactive services including:

- You cannot set an account for the service when running it as an interactive service (i.e. the `ServiceAccount` method will not be called if you return `True` for this method).
- If the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Windows` has the value **NoInteractiveServices** set to any non-zero value, the service will not be allowed to run interactively even if you return `True` from this method.

Microsoft discourages use of interactive services.

When a service is configured to interact with the desktop, the `GetInteractiveUser` method of the `IdwServiceCtl` interface is enabled (in Windows 2000/XP). Interactive services may also implement the `IdwEasyService2` interface to detect when an interactive user logs out.

ServiceAccount() As String

This method allows you to set an account other than `LocalSystem` for your service. Most services run in the `LocalSystem` account because this account has the necessary privileges to perform a variety of operations on a system, to log on as a different user, and to impersonate users or clients accessing the service via DCOM. However, the `LocalSystem` account does not have the ability to access most network resources.

If you wish your service to log on as a specific user, return the user account name as a result of this method as follows:

```
IdwEasyServConfig_ServiceAccount = "domain\userid"
```

Important Notes:

- This method will not be called if you returned `True` for the `IdwEasyServConfig_ InteractWithDesktop` method.
- The user account must have the privilege “Log in as a Service” enabled (set privileges using system administration tools).
- You can also change the service log on parameters using the system administrative tools.
- You must include the domain name, or “.” to indicate that the account is on the local system.

- The user account specified here can be overridden during installation using the -User command line option.

If you specify an invalid user ID or password, or the account does not have permission to log on as a service, an error will occur while the service is installed.

ServiceAccountPassword() As String

If you specify a service account, this method will be called to obtain the password for the user.

```
IdwEasyServConfig_ServiceAccountPassword = "user
password"
```

- The password specified here can be overridden during installation using the -Password command line option.
- The password is not tested for validity at install time.

ServiceDependencies() As String

Services are often dependent on other services running. It is especially important that Windows know about these dependencies for services that are configured to run when the system starts. You can use this method to specify a list of dependencies for your services. To do so, return as a result the list of services separated by semicolons. If you have services grouped, you can use the + symbol as a prefix for a group name (refer to MSDN for information on service groups). Use the short form name of the service to specify services in this list. The short form name of the service is located in the subkeys of the registry's \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services key.

The service framework is dependent upon the Remote Procedure Call Service (RPCSS), which is automatically added to the dependency list regardless of whether you specify it or not. Thus for most services you need not return any result for this method. Refer to the IdwEasyServConfig_AutoStart function for important information on setting the Service Dependencies when automatically starting a service on system start up.

To specify dependencies, use code such as this:

```
IdwEasyServConfig_ServiceDependencies =
"FirstService;SecondService"
```

ServiceProcessId() As Long

This method is used by the service framework to determine the process identifier for your component.

You should declare the `GetCurrentProcessId` in the `ServiceConfiguration` module using the following code:

```
Private Declare Function GetCurrentProcessId Lib  
"kernel32" () As Long
```

And return the process ID using the following code inside the `ServiceProcessId` method:

```
IdwEasyServConfig_ServiceProcessId =  
GetCurrentProcessId()
```

Normally, when your service runs as a compiled finished service, the service DLL will be mapped into the same process of the service executable, and the process ID will be the same. However, while debugging your service in the Visual Basic IDE, your service component will actually be in the IDE process space, not that of the service. This can have an impact on your ability to perform certain operations (as you will see later).

The service framework uses this method to determine if you are doing this type of debugging and is able to help you deal with the debugging process by copying certain objects between the two processes behind the scenes. This is especially important when you use background threads to wait on synchronization objects. If you do not return the process ID in this method, you will not be able to debug code that uses synchronization objects using the Visual Basic IDE. (Note: Background threads for synchronization objects are not supported on the Light Edition of the Toolkit).

Implementing the Service Class

The Service class is your primary class for implementing the functionality of the service. The object is created when the service begins to run, and is released when the service terminates.

Please refer to the MSDN documentation for services for a more in-depth explanation of the various service controls and possible responses.

The object may implement other methods than those defined by the IdwEasyService interface. The interface is implemented by the object by adding the following line to the class module:

```
Implements IdwEasyService
```

Summary:

Class name: Service

Instancing: Multi-Use

Implements: IdwEasyService

- Rules:
- All methods of the interface must be implemented.
 - Empty method declarations are sufficient for all methods (though such a service wouldn't be particularly useful).
 - Do NOT pass references to this object to clients outside of the service component.
 - Be sure to release all references to this object from other objects in your component before the service stops.
 - Do not use Friend type methods and properties – they do not work well in a multi-threaded environment.
- Privacy Issues:
- As a multi-use object, it could conceivably be created by other applications.
- Versioning:
- The WaitComplete method of the IdwEasyService interface is present, but not implemented, in the Light Edition of the Toolkit.

Optional: the Service object may also Implement the IdwEasyService2 interface. This interface, used by services that are configured to interact with the desktop, can be used to determine when a user logs off the system.

IdwEasyService Methods Relating to State Transitions

The IdwEasyService interface includes methods that are called from the service control manager. These methods all exhibit similar behavior, and can be discussed as a group.

Each of these methods has a single ControlObject parameter that exposes an interface called IdwServiceCtl. The ControlObject object, is an object exposed by the framework to allow your service to interact with the framework.

These methods reflect requests from the service control manager for the service to change its state in some way. The service control manager will expect the service to respond within a certain time, or will assume that the service has failed. Before these methods are called, the service framework notifies the system that the service will be done or will provide updated information within a time specified by one of the default timeout values set originally by the IdwEasyServConfig_DefaultTimes method in the ServiceConfiguration object.

If you expect your service to be finished with the operation within that time, you need take no further action beyond responding to the state transition as appropriate for your individual service.

If you would like additional time for the state transition, you can use the UpdateTransitionTime method of the ControlObject object to specify the amount of time you expect your service to take.

If you would like to delay the handling of this request (for example, if you are waiting for an external event from a background operation or other object to occur), you can return the amount of time in milliseconds that you would like to defer handling of the request. After that time, the method will be called again and you will have 500 ms to either request more time using the UpdateTransitionTime method, defer handling of the method again, or indicate completion of the state transition by returning zero.

OnContinue

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl) As Long

Called when the service control manager “Continue” request is received. This request will only be sent if the service is configured to accept pause and continue requests. The service framework notifies the system that the service is in SERVICE_CONTINUE_PENDING state before calling this method. After you return zero from this method, the framework will place the service in the SERVICE_RUNNING state.

Refer to the description under IdwEasyService methods relating to service control manager events for a description of how to use this method.

OnPause

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl) As Long

Called when the service control manager “Pause” request is received. This request will only be sent if the service is configured to accept pause and continue requests. The service framework notifies the system that the service is in SERVICE_PAUSE_PENDING state before calling this method. After zero is returned from this method, the framework will place the service in the SERVICE_PAUSED state.

Refer to the description under IdwEasyService methods relating to service control manager events for a description of how to use this method.

OnStart

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl) As Long

Called when the service control manager is about to start the service. The service framework notifies the system that the service is in SERVICE_START_PENDING state before calling this method. After you return zero from this method, the framework will place the service in the SERVICE_RUNNING state.

Refer to the description under IdwEasyService methods relating to service control manager events for a description of how to use this method.

OnStop

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl) As Long

Called when the service control manager “Stop” request is received. This request will only be sent if the service is configured to accept stop requests. The service framework notifies the system that the service is in SERVICE_STOP_PENDING state before calling this method. After you return zero from this method, the framework will place the service in the SERVICE_STOPPED state.

Refer to the description under IdwEasyService methods relating to service control manager events for a description of how to use this method.

OnShutdown

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl, StopPending As Boolean) As Long

This notification differs slightly from the others. Called when the service control manager “Shutdown” request is received. This request will only be sent if the service is configured to accept shutdown requests.

The service framework calls this method immediately upon receipt of a shutdown notification.

It is strongly recommended that you perform any cleanup operations and return a zero value from this method as quickly as possible.

You can delay the system shutdown by returning a time delay in milliseconds as a result of this method and setting the StopPending parameter to True (you MUST do both to delay shutdown). In that case, the service framework will place the service in the SERVICE_STOP_PENDING state and proceed to call this method again after the delay you specified. At that time, you can continue as with any of the other state transition functions, calling either UpdateTransitionTime or returning additional delay values.

Depending on your system configuration, the system may shut down regardless of what you do here. You can extend the time until the system shuts down all services by changing the setting of the WaitToKillService registry value located under the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet/Control registry key. The default timeout value is 20 seconds.

Version 2.0 of the toolkit provides earlier and more reliable detection of system shutdown, especially with interactive services.

Be aware that during system shutdown other services will be shutting down at the same time as yours. This means that critical services that your service may be dependent on may already be shut down by the time your OnShutdown method is called. These services may include the event logging service, messaging, transactioning, etc. You should therefore use additional error checking during the OnShutdown method to catch errors that might not occur under normal circumstances.

You should reduce to an absolute minimum the work done during the Shutdown event. If you find your design requires extensive work during shutdown, consider redesigning your application to store on disk or in the registry information about pending work to be done next time the service starts up.

Remember that in extreme situations (power loss, or major system errors), no shutdown notification will arrive. Also be sure to upgrade to the latest service pack of NT and 2000, since there are a number of outstanding bugs in earlier OS versions that cause shutdown notification to services to fail under certain conditions.

Refer to the description under IdwEasyService methods relating to service control manager events for a description of how to use this method.

IdwEasyService Methods Relating to other Service Control Manager Events

OnUserControlCode

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl, ByVal ControlCode As Integer)

Called when the service control manager “UserControl” request is received. This allows your service to receive control code (which have a value from 128 to 255) from the service control manager. The meanings of these control codes are defined by your service.

Refer to the description under IdwEasyService methods relating to Service Control Manger events for a description of how to use this method.

OnParamChange

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl)

(Available in Windows 2000/XP only. Present but not supported in the Light edition of the NT Service Toolkit.)

Services are encouraged to store any startup parameters in the registry at location HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services*service name*\Parameters.

If your service can reload its configuration settings from these parameters while running, you can specify that your service can accept Parameter change events (use the ServiceConfiguration_Controls-Accepted method, or ControlObject’s IdwServiceCtl Controls-Accepted property to specify which Service Control Manager events your service will accept).

On receipt of this method, you should reload your service configuration based on the current registry settings.

OnHardwareProfileChange
(ByVal ControlObject As
EASYSERVLib.IdwServiceCtl, ByVal ChangeType
As Long, ByVal ChangeData As Long) As
Boolean

(Available in Windows 2000/XP only. Present but not supported in the Light edition of the NT Service Toolkit.)

If your service wishes to be notified when the current hardware profile changes, you can specify that your service can accept Hardware Profile change events (use the ServiceConfiguration_ControlsAccepted method, or ControlObject's IdwServiceCtl ControlsAccepted property to specify which Service Control Manager events your service will accept).

Refer to the MSDN documentation for the WM_DEVICECHANGE message for information on the ChangeType and ChangeData parameters (ChangeType corresponds to the wParam value, ChangeData to the lParam value).

Return True from this function to deny the request for a hardware profile change.

OnDeviceEvent
(ByVal ControlObject As
EASYSERVLib.IdwServiceCtl, ByVal EventType
As Long, ByVal eventData As Long) As Boolean

(Available in Windows 2000/XP only. Present but not supported on the Light edition of the NT Service Toolkit.)

If your service wishes to be notified when a specified device or set of devices change, you can specify that your service can accept device change events. (Use the RegisterDeviceNotification method of the ControlObject's IdwServiceCtl interface to accomplish this.)

Refer to the MSDN documentation for the WM_DEVICECHANGE message for information on the EventType and eventData parameters (EventType corresponds to the wParam value, eventData to the lParam value).

Return True from this function to deny the request for a hardware profile change.

OnPowerRequest
(ByVal ControlObject As
EASYSERVLib.IdwServiceCtl, ByVal
APMMessage As Long, ByVal Flags As Long) As
Boolean

(Available in Windows 2000/XP only. Present but not supported on the Light edition of the NT Service Toolkit.)

If your service wishes to be notified when a power management event occurs, you can specify that your service can accept power management change events (use the ServiceConfiguration_ControlsAccepted method, or ControlObject's IdwServiceCtl ControlsAccepted property to specify which Service Control Manager events your service will accept).

Refer to the MSDN documentation for the WM_POWERBROADCAST message for information on the APMMessage and Flags parameters (APMMessage corresponds to the wParam value, Flags to the lParam value).

Return True from this function to deny the request for a power management related change.

IdwEasyService Methods Specific to the Service Framework

The following methods reflect features extensions provided by the framework to authors of services using this Toolkit.

OnTimer
(ByVal ControlObject As
EASYSERVLib.IdwServiceCtl)

The Service class is designed ideally to operate in a way similar to a form – as an event driven component. The framework provides a built in timer that calls the OnTimer method periodically while the service is running.

The timer is initially enabled with a 1 second interval.

You can set the timer value at any time using the Timeout property of the ControlObject object. Setting the TimeOut property to zero turns the timer off.

This method will only be called while the service is in the running state (it will not be called during state transitions).

The accuracy of this timer is not guaranteed and cannot be used for real time applications.

When this method is called, no further OnTimer events will be received until you return from the function.

Note: Do not allow the code in this event to take longer than the default Pause or Stop times set in the IdwEasyServ-Config_DefaultTimes method of the service configuration object. If calls to this method take longer than those times, and a Stop or Pause operation arrives during this method call, it is possible that the service control manager will decide that your service is not responding before you have a chance to receive and respond to the OnStop or OnPause methods.

WaitComplete

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl, ByVal ThreadID As Long, ByVal CompletionType As Long, ByVal ObjectIndex As Long)

(Method is present but not implemented in the Light edition of the NT Service Toolkit.)

This method is called when a background thread wait operation is complete. Refer to the section on Background Threads and Synchronization Objects for a detailed explanation of this method.

IdwEasyService2 Interface Methods

The IdwEasyService2 interface is optional and may be implemented by a service object that is configured to interact with the desktop.

OnLogout

(ByVal ControlObject As EASYSERVLib.IdwServiceCtl)

This method is called when an interactive user logs off the system.

Note: On some systems, this event may be raised more than once when a user is logging off the system.

IdwServiceCtl - The Service Control Object

So far the two interfaces that have been described, IdwEasyServConfig and IdwEasyService, have both been used by the service framework to communicate with your service component. It should be no surprise that communication is equally important in the other direction.

The service framework exposes a component called the ServiceControl object that implements an interface called IdwServiceCtl. This object is passed to your service component at various times, and you may hold a reference to it if necessary for your application as long as you release it before your service stops.

Summary:

Class Interface: IdwServiceCtl

Rules: You may hold a reference to this object if you release it before the service stops.

Do NOT pass references to this object to clients outside of the service component.

IdwServiceCtl Properties

The following properties are exposed by the service control object.

InstallParameters (String)

This property can be used to retrieve the string set via the -Params option during installation.

StartupParameters (String)

This property can be used to retrieve the startup parameters string set via the service control manager (using the control panel applet or programmatic startup).

Timeout (Long)

This value indicates the interval of the built in timer in milliseconds. Your service object's IdwEasyService_OnTimer will be called each time this interval expires. The initial value is set to 1000. You can set this value to zero to disable the timer. Refer to the description of the IdwEasyService_OnTimer method for more information.

ControlsAccepted (Long)

This property allows you to specify the service control manager events that your service should receive. This can be a combination of one or more of the following enumerated constants:

SvcStop = 1	Service accepts Stop commands.
SvcPauseAndContinue = 2	Service accepts Pause/Continue commands.
SvcShutdown = 4	Service accepts Shutdown.
SvcParamChange = 8	Service accepts parameter changes (Win2K only).
SvcHardwareProfile = 0x20	Service accepts hardware profile changes (Win2K only).
SvcPowerEvent = 0x40	Service accepts power events (Win2K only).

When the service is in the running, stopped or paused state, the system will be notified about changes to this property immediately. If the service is in a state of transition, the system will not be notified until shortly before the next state transition method is called, or (if changed during the state transition event) until you either return from the state transition, or use the UpdateTransitionTime method on the service control object to request additional time.

Refer to the description of the IdwEasyServConfig_ControlsAccepted method for more information.

IdwServiceCtl Methods

The following methods are exposed by the service control object.

UpdateTransitionTime (Timeout As Long)

During a state transition, you can call this method to request additional time to complete the state transition. The system expects your service to complete its state transition in a specified time or it will assume that an error has occurred in the service. The timeout parameter specifies the additional time requested in milliseconds.

Refer to the description of the IdwEasyService interface for additional information on timing during state transitions.

StopService

Warning! Avoid use of this method!

Normally services are controlled through the service control manager. In the unlikely event that a service must stop itself, it should do so using the service control manager API functions.

The StopService method is intended to allow your service to exit in as clean a way as possible if it has determined an internal failure condition so serious that it cannot continue. The method causes a hard stop and exit of the service with minimal cleanup.

SetWaitOperation

(Objects() As Long, bWaitAll As Boolean, Timeout As Long) As Long

(Method is present but not implemented in the Light edition of the NT Service Toolkit.)

This method is used to create a background thread and start a wait operation on one or more synchronization objects. Refer to the section on Background Threads and Synchronization Objects for a detailed explanation of this method.

ClientExecuteBackground

(Identifier As Long)

(Method is present but not implemented in the Light edition of the NT Service Toolkit.)

This method is used to start a background execution on a client object identified by the client identifier specified by the identifier parameter. Read the section on exposing objects through COM and DCOM for additional information on this method.

ClearWaitOperation

(WaitThreadId As Long)

(Method is present but not implemented in the light edition of the NT Service Toolkit.)

This method is used to terminate a wait operation in progress. Refer to the section on Background Threads and Synchronization Objects for a detailed explanation of this method.

GetInteractiveUser (Domain As String) As String

(Method is present but not implemented in the Light edition of the NT Service Toolkit. Available in Windows 2000/XP only.)

This function returns the user name of the currently logged on user. The Domain parameter returned will be set to the name of the domain for the currently logged on user. If the user does not belong to a domain, it returns the name of the current computer in this parameter. This method is only enabled for services that are configured to interact with the desktop.

This function uses current system security settings to determine the interactive user. It is not guaranteed to work in all system configurations.

RegisterApplicationObject (AppObject As Object)

(Method is present but not implemented in the Light edition of the NT Service Toolkit.)

This method allows you to set an object to be the shared application object for your service. The service framework will make this object accessible to all clients of the service, making it possible for them to communicate both with the service and with each other.

Refer to the section on Exposing Service Objects through COM and DCOM for a detailed explanation of this method.

RegisterClientObjectName (ClientObjectName As String)

(Method is present but not implemented in the light edition of the NT Service Toolkit.)

This method allows you to specify the name of the object that the framework should create when users request client objects. If you do not specify a client object name using this method, your service will not expose objects through COM. If you do specify an object name, the service will expose the RunningService object which will inherit the public methods and properties of the object whose name you specify here.

Refer to the section on Exposing Service Objects through COM and DCOM for a detailed explanation of this method.

RegisterDeviceNotification (NotificationFilter As Long) As Long

(Method is present but not implemented in the Light edition of the NT Service Toolkit. Available in Windows 2000/XP only.)

This method wraps the RegisterDeviceNotification API call that allows you to obtain notifications when devices are added, removed or reconfigured on a system. The NotificationFilter parameter should be a pointer to a block of memory containing a description of the device or type of device to detect (see the API declaration description in MSDN for details on the format of this memory block). Refer to the book “Dan Appleman’s Win32 API Puzzle Book and Tutorial for VB Programmers” for detailed information on working with arbitrary memory blocks and pointers from Visual Basic.

The method automatically sets up events to be processed by the IdwEasyService_OnDeviceEvent method of your service object.

The return value is a handle to the notification object which can be used to unregister the device notification using the UnregisterDeviceNotification method. The method will raise an error if the operation fails.

You are encouraged to call UnregisterDeviceNotification when your service stops. The framework will attempt to do so if you do not.

Do not use the UnregisterDeviceNotification API with handles returned from this method.

UnregisterDeviceNotification (DeviceNotification Handle As Long)

(Method is present but not implemented in the Light edition of the NT Service Toolkit. Available in Windows 2000/XP only.)

This method terminates device notification. The DeviceNotificationHandle parameter is the handle to the device notification object returned by the RegisterDeviceNotification method.

ReportEvent

(EventType As enumEventReportTypes, EventString As String, BinaryData() As Byte)

This method adds an event to the application event log.

The enumEventReportTypes parameter can be one of the following three values:

svcEventlogError = 1	Records the event as a service error.
svcEventlogWarning = 2	Records the event as a service warning.
svcEventlogInformation = 4	Records the event as an information notification only.

The EventString string is text that is written directly into the event log for the event. The BinaryData parameter is a byte array that contains any arbitrary data you wish to include with the event.

More sophisticated event logging is possible using the ReportEvent2 function or API techniques and language independent message files for those who require advanced logging or the ability to categorize events, or filter based in types of events.

ReportEvent2

(Source As String, EventType As enumEventReportTypes, Category As Integer, EventId As Long, [EventString], [BinaryData])

This method adds a detailed event to the application event log, for use with Desaware's Event Log Toolkit or other custom event sources.

The Source parameter contains the Event source name (assumed to be on the local system).

The enumEventReportTypes parameter can be one of the following three values:

SvcEventlogError = 1	Records the event as a service error.
SvcEventlogWarning = 2	Records the event as a service warning.
SvcEventlogInformation = 4	Records the event as an information notification only.

The Category parameter contains the category number of the event.

The EventId parameter contains the event identifier (including the facility, severity and event code number).

The EventString parameter is optional and is a variant containing either a single string or a string array (zero based) containing the event string parameters.

The BinaryData parameter is optional and is a variant containing a byte array (zero based) containing any binary data.

Trace

(TraceString As String, TraceLevel As Long)

This method adds the specified string to the current trace log. These messages will be combined with any messages produced by the framework.

The TraceLevel parameter is used to specify the severity of the error, with 1 being most severe and 4 least severe. Whether the message is actually logged will depend on the current trace level as set in the service's profile (.ini) file. Refer to the section on Testing and Debugging for more information on the tracing and logging capabilities of the framework.

Using the Service Configuration Program

The Desaware NT Service Configuration Wizard program creates a custom service executable specifically for your Visual Basic project. This Wizard guides you through a series of steps requesting information regarding your Visual Basic service. The following describes each step.

Service Executable Name

This is the first step in the Wizard. Enter the name of the service executable file. It can be any name you choose. If you select an existing service executable file, this wizard will extract the service information from that executable and initialize the remaining steps of the wizard with that information. You can use the Browse button to navigate your file system to specify a file name for your service executable.

Tip: You can create a service executable file that contains some default information (such as version information) for your company or product and use it as a template file. Each time you need to create a new service executable, select the template file to initialize this Wizard with the version information, etc., then change the service executable file name to the name you want to give for your service.

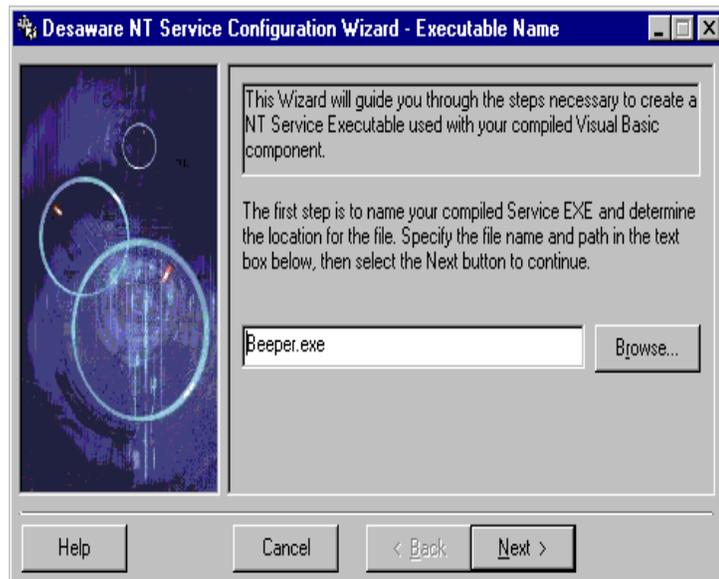


Figure 2
NT Service Configuration Wizard
Assigning Executable Name & Location

Project Name

Enter the project name of the Visual Basic DLL that contains the Service and ServiceConfiguration objects with which your service will communicate. If you decide later that you will want to change the project name, you will have to run this configuration program again.

It is important that you should choose a component name that is unique – duplication of project names used by this framework can cause services to fail to work properly. We recommend including your company name or initials in the name. For example: most Desaware components include the prefix “dw”. The project name length can be up to 18 characters (due to the 39 character limitation for the combined project and class name, and the requirement that the “ServiceConfiguration” class name needs to be exposed).

Version Information

Enter the version resource information for your service executable file. The Service Configuration Wizard writes the same version resource information to your service executable as Visual Basic. You must enter information in the “Company Name” and “File Version” fields. The “File Version” field must contain a valid version number in “#.#.#.#” format, for example “1.0.0.1”. If you select an existing service executable file to compile, its file version will automatically be incremented by 1 revision where revision is the fourth version number field in the version format.

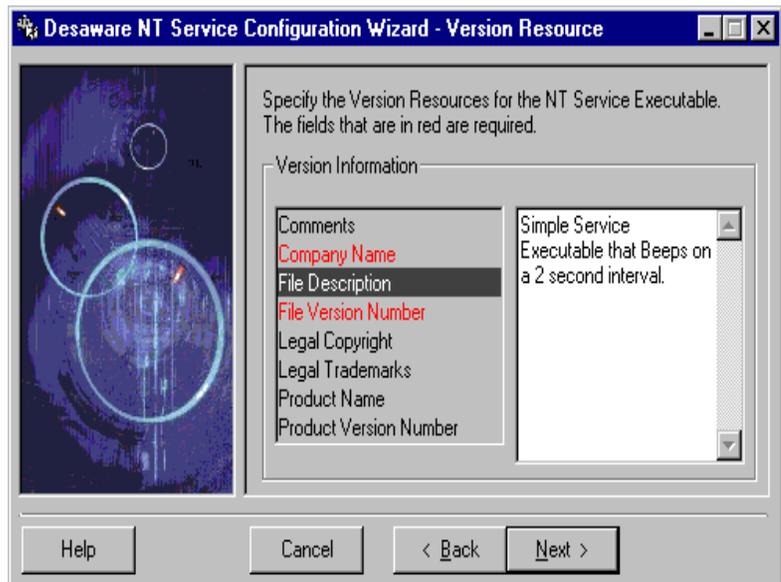


Figure 3
NT Service Configuration Wizard
Specifying Version Resources

Thread Count

(Not available in Light edition)

Enter the number of threads to allocate from the Thread Pool. If your service will expose COM objects for use by clients using the service, those objects will be created on a thread pool so that their operation will not interfere with the main service. The thread count must be in the range from 1 to 32, the default is set to 4.

Create VBR File

(Not available in Light edition)

Enter the description of the service object for your service. If your service will expose objects that clients can access through DCOM, you'll need to create a VBR file which is used by the clireg32 application to create the necessary registry entries for accessing your service objects remotely. The Service Configuration Wizard will create a VBR file for you automatically on request. The VBR file will be created in the same directory and have the same base name as your service executable.

Compile Executable

The Service Configuration Wizard is now ready to compile your service executable file. Select the *Next* button to start compilation, or the *Back* button to make any changes.

Compile Completed

The Service Configuration Wizard has finished compiling your service executable. This step displays whether the compilation was successful or not. After a successful compilation, you need to install your service executable before you can run it. If you would like to run your service executable in our Service Simulator, you need to register your service. Be sure that your Visual Basic project containing the Service objects is already registered prior to installing or registering your service executable. You can use the Service Executable Launcher utility to install, uninstall, register, and unregister your service executable.

Running the Service Configuration Wizard in Batch Mode

(Not available in Light edition)

The Desaware NT Service Configuration Wizard program supports batch commands so you can build your Service Executable as part of an automated build script. You can use the batch command line switches to update an existing Service Executable (or create a new Service Executable based on an existing one). The following describes the batch command switches.

Command Switches

- | | |
|--|---|
| <code>/batchbuild</code> | This switch is necessary in order to run the NT Service Configuration Wizard in batch mode. |
| <code>/infile=<i>filename</i></code> | This switch is necessary. The <i>filename</i> variable represents the existing Service Executable upon which the newly compiled Service Executable will be based. The newly compiled Service Executable will use the same Service Project, retain the same Version Resource, Thread Pool, and DCOM settings as the existing file. The filename variable should also include the path. |
| <code>/outfile=<i>filename</i></code> | This switch is optional. The <i>filename</i> variable represents the path and file name for the newly compiled Service Executable file. If omitted, the newly compiled Service Executable file will replace the file specified by the <code>/infile</code> switch. |
| <code>/fileversion=<i>x.x.x.x</i></code> | This switch is optional. This switch sets the file version number of the newly compiled Service Executable file to the specified version number. If omitted, the file version number will not change. If the <code>/fileversion</code> switch is specified without any version number, the file version number will increment by 0.0.0.1 from the file version number of the file specified by the <code>/infile</code> switch. |

/productversion=x This switch is optional. This switch sets the product version number of the newly compiled Service Executable file. If omitted, the product version number will not change. If the */productversion* switch is specified without any other string, it will increment by 0.0.0.1 from the product version number of the file specified by the */infile* switch. If the */productversion* switch specifies a valid version number, it sets the product version number of the newly compiled Service Executable file to that valid version number. If the */productversion* switch is set to “fileversion”, it sets the product version number of the newly compiled Service Executable file to the same as the file version number.

/logfile=filename This switch is optional. This switch sends the batch results from the currently executing Desaware NT Service Configuration Wizard program to the specified log file.

The filename parameter should include both the path and file name of the path. The results will be appended if the file already exist. If omitted, the batch results will default to current folder's NTServiceWizard.log file.

Here are some sample batch command lines and their results.

The following batch command line rebuilds the existing beeper2.exe file. The newly compiled file's file version number is set to 2.0.1.0. The newly compiled file's product version number is set to the same as the file version number (2.0.1.0), and the build results are logged to the beepererror.log file.

```
NTServiceWizard.exe /batchbuild /fileversion=2.0.1.0
/infile=c:\ntservtk\samples\beeper\beeper2.exe
/logfile=c:\ntservtk\samples\beeper\beepererror.log
/productversion=fileversion
```

The following batch command line rebuilds the existing beeper2.exe file as beeper3.exe. The newly compiled file's file version number and product version number are incremented by 0.0.0.1. The build results are logged to the NTServiceWizard.log file.

```
NtServiceWizard.exe /batchbuild /fileversion /productversion  
/infile=c:\ntservtk\samples\beeper\beeper2.exe  
/outfile=c:\ntservtk\samples\beeper\beeper3.exe
```

Using the Service Executable Launcher Program

The Desaware NT Service Executable Launcher program helps you install, uninstall, register, and unregister your service executable. You should have already registered your Visual Basic project containing the Service objects prior to attempting any of these commands. To use the Service Executable Launcher, just drag and drop your service executable from Windows NT Explorer onto any of the command buttons. The command described on each command button will be implemented on the dropped files. You may also drop the files on the text box or use the Browse button to select one or more files. This will put the file names and paths into the text box. Afterwards, you may click on any of the command buttons to implement that command on all of the files in the text box.

To install or register your service with additional parameters, just fill in the appropriate User, Password, or Params text boxes prior to running the Install Service or Register for Simulator commands. **NOTE:** If you had successfully installed a service and you make any service configuration changes, you must uninstall the service first and then install it again before those changes to take effect.

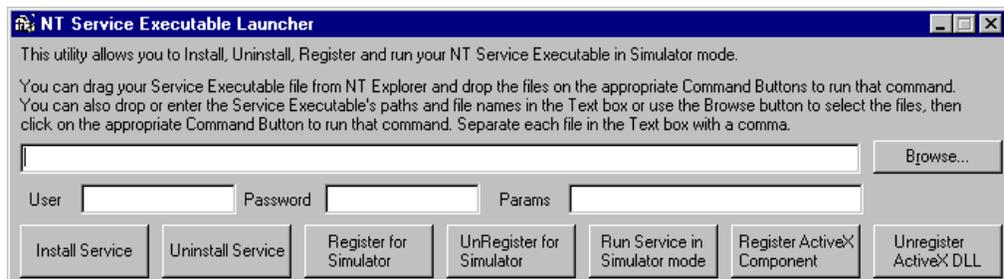


Figure 4
NT Service Executable Launcher Utility

Background Threads and Synchronization Objects

(Note: Background threads are not implemented on the Light edition of the NT Service Toolkit.)

Windows supports a wide variety of synchronization objects. These are objects that allow you to wait for something to occur. This can vary from waiting for a process or thread to end, to waiting for a timer, to waiting for a change to the registry or a file. You can also wait on objects such as mutexes, events and semaphores that are used to synchronize threads and processes. Each of these objects can be in a non-signaled or signaled state.

An in-depth discussion of synchronization objects is beyond the scope of this manual. You can find an extensive discussion of synchronization objects and the API functions required to use them in Visual Basic in the book “Dan Appleman’s Visual Basic Programmer’s Guide to the Win32 API”. Information on synchronization objects can, of course, be found in the MSDN library. You will use standard Win32 API functions to create and free the Windows synchronization objects that you use with these methods.

The Win32 API includes functions that can wait for a synchronization object to be signaled. There are two ways to use these functions. The best way is to suspend the thread until the object is signaled. This approach is extremely efficient because it uses almost no system resources. Unfortunately, it does have the side effect of freezing the thread – a serious problem if your application has only one thread. The alternate approach typically used by Visual Basic programmers is to use the wait functions with very short timeouts, and check afterwards if they returned due to an event being signaled or due to a timeout. In effect, you end up polling the objects – a very inefficient approach.

One of the most important tasks often handled by services is to monitor system events. Thus being able to wait efficiently for a synchronization object to be signaled is extremely important. The service framework has the ability to create background threads for this purpose.

Synchronization object handles are associated with individual processes, which adds a level of complexity when debugging services using the Visual Basic IDE (in which your component is in a separate process from the service framework). The framework detects this situation by examining the process identifier returned from the `IdwEasyServConfig_ServiceProcessId` of your `ServiceConfiguration` object and automatically duplicates the objects into its own process space to enable the background thread functionality even when debugging using the Visual Basic IDE.

Methods Used to Implement Background Threads

The following methods are used to control background thread in the NT service framework.

Control Object (IdwServiceCtl Interface)

The following two methods are exposed on the `IdwServiceCtl` interface of the control object, allowing your service to create or terminate background threads:

```
SetWaitOperation(Objects() As Long, bWaitAll As Boolean, _ Timeout As Long) As Long
```

This method takes an array of synchronization objects, creates a background thread, and performs an efficient wait for a specified wait condition.

The `Objects()` array is an array of `Long` variables that you define using the `Dim` statement. The array should be declared for the exact number of object handles you will use. Thus you would specify a dimension of one less than the total number of objects. You then assign each of the object handles to entries in the array. For example: to wait on a single object, you would use the following code:

```
Dim ObjectList(0) As Long  
ObjectList(0) = yourhandle ' Handle to your  
synchronization object  
ThreadId =  
ControlObject.SetWaitOperation(ObjectList(), _  
False, timeoutvalue)
```

The `bWaitAll` parameter determines whether you want to wait until all of the objects are signaled, or until any one of the objects is signaled. When `True`, the function will wait until they are all signaled (in which case they must all be in the signaled state at the same time for the wait condition to be satisfied – if one becomes signaled and then unsignaled and then the rest of the objects become signaled, the wait condition will not be satisfied).

The `Timeout` parameter allows you to set an overriding timeout value in milliseconds. The wait condition will be automatically satisfied when the timeout expires regardless of the state of the objects.

This method returns the thread identifier which you can later use to identify the background thread.

The number of background threads you can create is not limited by the framework, but may be limited by the system.

When the wait condition is satisfied, the `IdwEasyService_WaitComplete` method will be called on the `IdwEasyService` interface of your service object.

If an error occurs in setting the wait condition (such as specifying an invalid synchronization handle), it will be reported by the framework calling the `IdwEasyService_WaitComplete` method with the `CompletionType` value set to `-1`. The error cannot be reported immediately because it does not actually occur until the new thread is created and the wait operation is attempted.

```
ClearWaitOperation(WaitThreadId As Long)
```

This method is used to terminate a wait operation in progress. The `WaitThreadId` parameter is the thread identifier obtained previously using the `SetWaitOperation` method.

You do not need to use this method on threads that have terminated due to a successful wait or timeout (i.e. those threads that have called the `IdwEasyService_WaitComplete` method on your service object.)

It is recommended that you clear any pending wait operations when you are notified that the service is about to stop. However, if you fail to do so, the framework will attempt to clean up for you. However, in doing so it will not release any of the objects that you are using which could result in a system resource leak (depending upon the objects in use).

Service Object (IdwEasyService interface)

The following method is exposed by your service object to receive notifications when wait conditions are satisfied:

```
IdwEasyService_WaitComplete(ByVal ControlObject As  
EASYSERVLib.IdwServiceCtl, ByVal ThreadID As Long,  
ByVal CompletionType As Long, ByVal ObjectIndex As  
Long)
```

The ControlObject parameter is a reference to the service control object from the service framework.

The ThreadID parameter identifies the background thread whose wait condition has been satisfied.

The meaning of the CompletionType and ObjectIndex parameters depends upon whether you are waiting for all objects or one object to be signaled (i.e. whether the bWaitAll parameter to the SetWaitOperation method is True).

If you are waiting on one object only...

The CompletionType parameter will be one of the following four values:

- 1 Indicates the wait operation failed.
- 0 Indicates that the wait condition was satisfied (at least one object was signaled). The ObjectIndex parameter is the entry in the array of the object that satisfied the wait condition.
- 1 Indicates that the wait condition timeout expired.
- 2 Indicates that a mutex object was abandoned. The ObjectIndex parameter is the entry in the array of the abandoned mutex that satisfied the wait condition.

If you are waiting on all objects...

The CompletionType parameter will be one of the following four values:

- 1 Indicates the wait operation failed.
- 0 Indicates that the wait condition was satisfied (all objects were signaled).
- 1 Indicates that the wait condition timeout expired.
- 2 Indicates that at least one mutex object was abandoned, and all other objects were signaled. The ObjectIndex parameter is the entry in the array of an abandoned mutex.

After this method is raised, the background thread used to perform the wait operation is automatically terminated.

Exposing Service Objects through COM and DCOM

(This feature is not supported under the Light edition of the Toolkit.)

One of the most important features of a service is the ability to act on behalf of a client. And the best way for clients to obtain access to a service is using COM or DCOM to obtain a reference to an object exposed by the service.

The Service Framework Object Architecture

There are two types of objects you can expose through your service component.

1. Objects that run in the primary service thread.
2. Objects that run within a thread pool provided by the service.

Let's consider both types of objects.

Objects That Run on the Primary Service Thread

When an object runs within the primary service thread, it shares memory with the Service object. This means that it is very easy to exchange data with the main body of the service. More important, the object can be shared among all of the clients. The disadvantage of this type of object is that all method and property calls to the object are on the primary service thread – thus a long operation performed by a method on this object can impair the performance of the entire service.

The service framework allows you to register an object from your component to serve the role of an “application” object. You create an instance of the object in your main Service object, and register it by calling the RegisterApplicationObject method on the ServiceControl object (IdwServiceCtl interface). The framework holds a reference to this object until the service terminates. Since the object was created by your service object, it runs in the same thread as the service object and shares its memory space. This object should be marked as Public-Noncreatable so that it will not be created outside of the service. The object can have any name – our samples use the name “Application” by convention. Note that in this it differs from the main classes you provide, “Service” and “ServiceConfiguration” in which the framework expects your classes to have a specific name.

Your application object is typically used to expose functionality that controls the entire service or is otherwise global to the service. It may also be used by client objects (which you'll read about shortly) to exchange data among clients.

Objects That Run Within a Thread Pool Provided By the Service

The application object is fine for providing functionality that is global to a service and consists of short operations, but is terrible for general support of clients. For a service to support clients properly, you need a different set of features, specifically:

- Long client operations should not impact the overall performance of the service.
- Long client operations should have minimal impact on other clients.
- The service should be able to act on behalf of individual clients based on their identity and security context.

These goals are accomplished by having client objects run on a different set of threads from the primary service. The service configuration program allows you to set the number of threads in a thread pool that is made available to all client objects. That way, if a client performs a long operation, at worst it will block another client in the same thread. The service itself, and all clients on other threads, will continue to run. Distributing the client load among multiple threads is the standard mechanism by which services achieve a high degree of scalability as the number of clients increases.

You define a client object by creating a class marked as `MultiUse` in your component, and registering the name of the class with the `ServiceControl` object using the `RegisterClientObjectName` method.

The RunningService Object

The service framework always exposes a COM object named `RunningService`. Thus, if you configured your service executable with the name *MyService*, the framework will expose a public object called *MyService.RunningService*.

If you have not registered an application object, and have not registered a client object name (i.e., you are not exposing any COM objects from the service), any attempt to create the object *MyService.RunningService* will fail with the error “ClassFactory cannot supply requested class”.

If you have registered an application object, any legal attempt to create the *RunningService* object will succeed, and retrieve an object with a single automation method called “GetAppObject”. This method returns a reference to the object you registered earlier. *GetAppObject* returns an error if the service is not in the running state.

If you have registered a client object name, any legal attempt to create the *RunningService* object will succeed, and retrieve an object that has both the “GetAppObject” method described earlier, and every public method and property that you defined in your client object class (yes – the methods you defined in your client are added to the services *RunningService* object).

Your client object will be created on the thread pool described earlier, thus will run in the same thread as the *RunningService* object – not the main *Service* object. As such, your client object will not share global memory space with the other objects in your service component.

All methods and properties of the client object are accessed through the *IDispatch* (automation) interface and are thus late bound. This means that references to the objects should be defined “As Object” rather than as the specific object type (the performance impact of using late binding in this case is negligible compared to the marshalling overhead that you are going through with COM or especially DCOM).

The service must be in the running state for a client to obtain a *RunningService* object for the service. The client will not be able to obtain an object in any other state (including the Paused state). If the client is holding a reference to the *RunningService* object, all property and method access to your object will continue to work while the service is paused. It is up to you to decide how to handle incoming client requests while the service is paused.

If your client object raises any runtime errors during method or property access by the client, those errors will be reflected through the *RunningService* object to the client.

Creating the Application Object

Creating an application object for your service is almost trivial. Simply do the following:

1. Add a new class to your component, typically named “Application” (though it can be any name).
2. Set the Instancing for the component to “2 – Public not Creatable”.
3. In your service object, create an instance of the application object, typically using code such as:

```
Private WithEvents appobject As Application
```

4. In the `IdwEasyService_OnStart` method for your service object, register the application object using code such as:

```
Call ControlObject.RegisterApplicationObject(appobject)
```

That’s all there is to it.

Of course, there are some rules you should follow.

Reference Counting and the Application Object

There are two issues to watch for with regards to the Application Object that relate to reference counting and making sure that your object is properly freed when the service stops.

The first, and easy issue, is to make sure your Service object frees the application object when the service stops. The way to do this should be familiar – simply set any references to the object to “Nothing” during the `IdwEasyService_OnStop`, `IdwEasyService_OnShutdown` method calls, or class terminate events. We recommend freeing all references during the `OnStop` and `OnShutdown` method calls because it is a much more predictable time to perform cleanup tasks. By the time the class terminate event fires, you can’t be sure which other objects in your component have been freed by Visual Basic, so access to methods of other objects is potentially risky if your object model is complex.

A second, and trickier issue, has to do with accessing the Service object from the Application object.

Accessing the Service Object from the Application Object

You may need to access methods and properties of your Service object from the Application object. A typical case is where the Application object wishes to start a wait operation using the Service Control object using a reference held by the Service object. There are three approaches you can use to handle this situation:

1. Use a global variable to hold a reference to the Service object.
2. Store a reference in the Application object to the Service object.
3. Use an event from the Application object to obtain a reference to the Service object.

The first two of these approaches have the well known problem of circular references. How can the Service object ever be freed if the Application object or a global variable are holding a reference to the service object?

Well, the good news is that you are running in a service framework that is doing a lot of work for you behind the scenes. For one thing, it sends you notification when the service needs to stop so you can clear those references and thus allow the Service object to free itself. Second, the framework will ultimately force your objects to terminate when it kills the thread that they live on – but that’s not a clean solution and could cause memory exceptions depending upon what your service is doing.

Clearing references during the OnStop and OnShutdown method calls is a fine solution, and perfectly reliable. However, a more elegant solution is to use the following event mechanism to acquire a reference to the Service object only when it is needed.

In the application object, create the following event:

```
Event GetServiceObject(svc As Service)
```

Any time you need a reference to the Service object, raise the event.

In the Service object, declare you application object variable with events as follows:

```
Private WithEvents appobject As Application
```

And add the following event handler:

```
Private Sub appobject_GetServiceObject(svc As _  
Service)  
    Set svc = Me  
End Sub
```

Event handlers use weak binding – meaning that applications that raise events don’t hold a reference to the objects that receive those events. Since the application object does not hold a reference to the Service object, the service object will be freed normally by the framework when the service ends.

The framework supports having the application object raise events to the service object. It does not support raising events to clients that have references to the Application object. If you need a communication channel back to clients, we recommend doing so through the Client object, not directly from the application object. In theory, you can use OLE callbacks from the application object to clients, however, this could seriously impact the performance of your service (since it would block your primary service thread until the call returns), and may have other side effects of which we are not aware.

As will be described later, the framework does additional work behind the scenes to help clean up after clients who are using your application object, even if they are ill-behaved and holding references to the object. So while you should definitely pay close attention to circular references in your own application, you don’t have to worry too much about what your component’s clients are doing with those objects.

Creating the Client Object

Creating a client object for your service is almost as easy as creating the application object. Simply do the following:

1. Add a new class to your component, typically named “Client” (though it can be any name).
2. Set the Instancing for the component to “5 – Multiuse”.
3. Implement the `IdwServiceClient` interface using the following code:

```
Implements IdwServiceClient
```

4. Add code for the `IdwServiceClient` methods (description follows). As with other interfaces, the default operation if you just add an empty method declaration is sufficient for most applications.
5. In the `IdwEasyService_OnStart` method for your service object, register the client object using code such as:

```
Call ControlObject.RegisterClientObjectName _  
("clientobjectname")
```

where *clientobjectname* is the name of your client object class

That's all there is to it.

Again, there are some rules you should follow, and some subtle issues to be aware of. They will be described shortly.

The IdwServiceClient Interface and the Unusual Life of Client Objects

The `IdwServiceClient` interface must be implemented by the object that you wish to use as your client. In order to clearly understand how it works, let us first take a close look at some unusual problems that occur when you expose objects from a service.

Reference Counting and the Client Object

In a normal component that exposes objects, the lifetime of the component is dictated by the life of the objects. Once all of the clients release their objects, the component shuts itself down and unloads.

In a service, the lifetime of the component is dictated by the service control manager and is independent of the life of any objects it exposes!

This has profound ramifications.

You've already read that a service will not allow objects it exposes to be created unless the service is actually running.

But consider the flip side – this also means that if you stop the service, any objects it exposes will stop working for clients!

The service framework handles this by forcibly disconnecting any clients that are using objects as soon as the service stops. Those clients will receive “Object disconnected” error messages if they attempt to access the client object once the service stops.

But there's more.

Consider how your service will interact with client objects.

Obviously, you'll want your client objects to be able to communicate with your primary service object. Keep in mind that your client object is running on a thread pool and thus will not have access to any global variables in your component. This suggests that your client object might want to hold on to a reference to the primary service object. How does it get a reference to the primary service object? The service framework calls your client object's `OnConnect` method (on the `IdwServiceClient` interface that it implements) and passes it a reference to your service object.

But at the same time, your service object might want to keep track of clients using the service. To do so, it needs a way to gain a reference to the client object and possibly hold a reference to the client object. The best way for it to gain a reference to your client object is by having your client call a method on the service object as soon as it receives the `OnConnect` method call. Your service object can store references to all of the clients in a collection.

But if the client object holds a reference to the service object, and the service object holds a reference to the client object, you have a classic circular reference. Neither object will ever be freed. Neither `Terminate` event will ever fire.

Fortunately, the service framework, combined with the fact that services will stop when instructed to do so regardless of references, presents an elegant solution to this problem.

Figure 5 illustrates the actual object model used when you expose a client object.

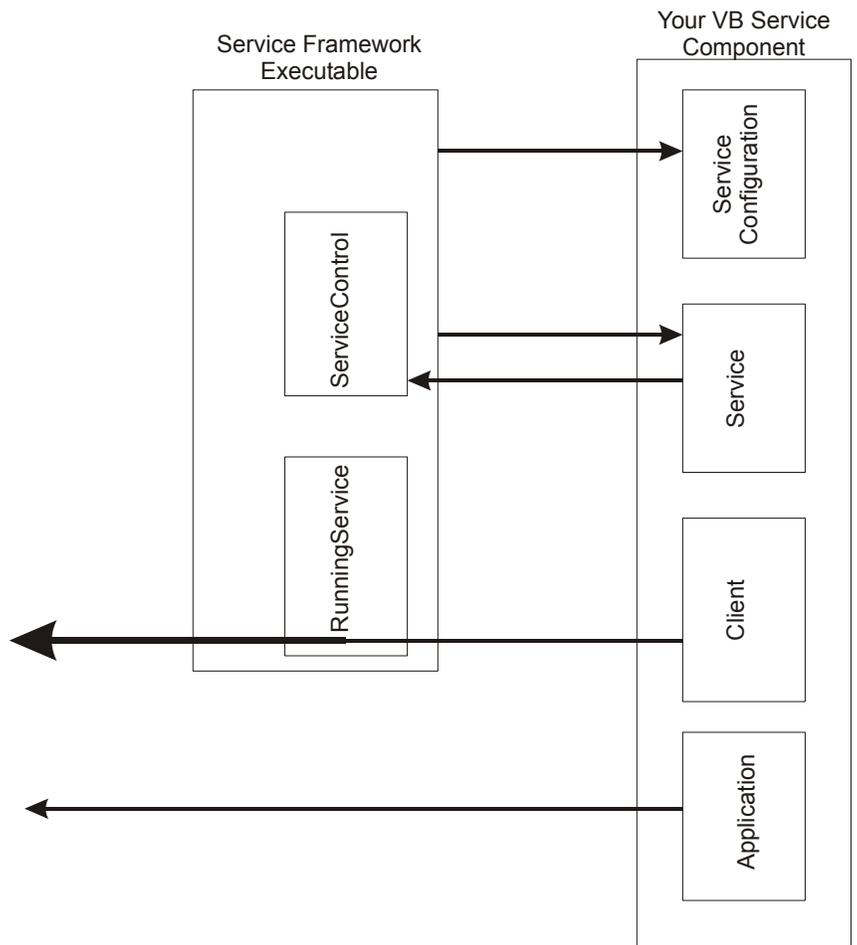


Figure 5
The Client Object Model

Even though the `RunningService` object exposes all of the public methods and properties of your client object to the client, from an object hierarchy point of view, its lifetime is independent of your client object. The `RunningService` object can be released in two ways: when the client releases its last reference, or when the service stops and forcibly disconnects the client. In this case the `RunningService`

object will know that it should terminate – but how does it help your client object, which is trapped in a circular reference?

Simple – before the RunningService object self-destructs, it calls the OnDisconnect method. During this method call you should break any circular references and thus allow your client object to be freed.

Here is a simple example that illustrates one way to handle this situation.

In your service object, add the following code:

```
Dim ClientCollection As New Collection

Friend Sub RegisterClient(obj As Client)
    ClientCollection.Add obj
End Sub

Friend Sub UnregisterClient(obj As Client)
    Dim tobj As Client
    Dim x As Long
    For x = 1 To ClientCollection.Count
        If ClientCollection.Item(x) Is obj Then
            ClientCollection.Remove (x)
            Exit Sub
        End If
    Next x
End Sub
```

In your client object, implement the following code:

```
Private Sub IdwServiceClient_OnConnect(ByVal
ServiceObject As Object)
    Dim svc As Service
    Set svc = ServiceObject
    svc.RegisterClient Me
End Sub

Private Sub IdwServiceClient_OnDisconnect(ByVal
ServiceObject _
As Object)
    Dim svc As Service
    Set svc = ServiceObject
    svc.UnregisterClient Me
End Sub
```

Figure 6 illustrates the relationship between the RunningService object and your client object.

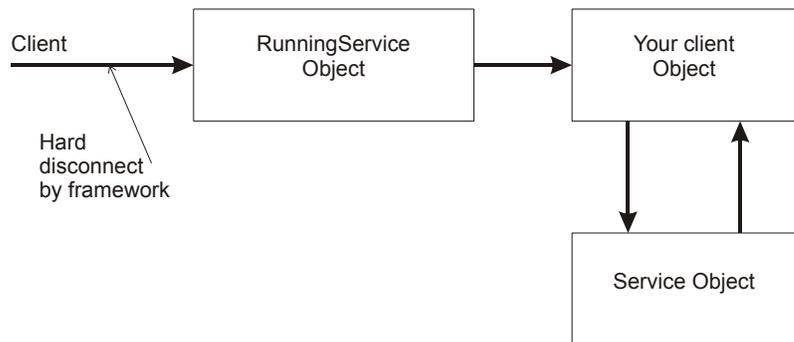


Figure 6
The Relationship Between the RunningService Object
and Your Client Object

The extra step of setting the ServiceObject object to a variable of type Service is necessary to call the functions in the service object because they are defined as Friend functions (not visible outside of the component). This is probably overkill, since you shouldn't be exposing your service object outside of the component anyway.

By holding on to references to your clients, you can also notify them when the service is about to stop, and keep the service running until each one has completed any necessary cleanup (using the techniques for deferring the stopping of a service described earlier).

What if your client objects run independently and you do not need to hold a reference in the service? The service framework provides an OnStop method on the IdwServiceClient interface that notifies the client object that the service is being stopped. Try to keep operations in this method as short as possible, since code in this function will delay the unloading of the service executable after the service has formally stopped.

The following functions are defined on the IdwServiceClient interface:

OnConnect

(ByVal ServiceObject As Object, ByVal ObjectIdentifier As Long)

This method is called after the class Initialize event, after a client has connected to the object. During this method you can store a reference to the primary service object, and call methods on that object to register the client with the service. The ObjectIdentifier parameter is a value that identifies the particular client object. This value can be used with the service control object's ClientExecuteBackground method to cause an asynchronous call to the ExecuteBackground method.

OnDisconnect

(ByVal ServiceObject As Object)

This method is called after a client has disconnected from the RunningService object. You should free all references to your client object when this is called – especially references held by the primary service object.

OnStop

(ServiceObject As Object)

This method is called when the service is about to stop. Perform any necessary cleaning here and try to avoid long operations.

Note: It is your responsibility to defer the service from stopping until all of the client's OnStop methods are called if you design your object such that it must receive this method call. If you return immediately from the IdwEasyService_OnStop method (in your service object), the service will shut down immediately and you are not guaranteed to receive this method in each client object.

The framework works this way because it is possible there is no way for the framework to know how long it will take for the IdwServiceClient_OnStop method call to return. Since the Service object controls termination of the service, the framework cannot guarantee that this method will be called before the service terminates.

Refer to the RemoteUser sample application for an illustration of one approach for handling this situation.

ExecuteBackground()

If you call the ClientExecuteBackground method of the Service control object using the object identifier you received from the IdwServiceClient_OnConnect method, you will start an asynchronous call to this method.

In other words, the call to the ClientExecuteBackground method will return immediately and you can return from the current function call. During the next opportunity for events to be processed in the thread (meaning you've either returned from the function call, called DoEvents, or raised an event), this method will be called by the service framework.

This feature allows your client to start operations on the client object and return immediately. The background operation occurs on the thread of the client object, which does not interfere with the primary service thread.

Additional Application and Client Object Issues

Exposing COM objects from a service is complex in general, given the multithreaded nature of services. We've made every effort to keep it as simple as possible, but there are issues you should understand that are natural consequences of this architecture. Some of these issues relate to NT services, some to COM, some to Visual Basic, and some to the service framework. Many of them will be familiar to you from your regular programming efforts, but may not be intuitive when applied to services.

Global Variables

Client objects should NEVER access global variables, only variables defined in the class module itself. Not only don't they have access to the main service thread's global variables, because they are on a thread pool, you'll never know which clients might be sharing global variables! (All objects on a thread in Visual Basic share global variables.)

Service State and the Client and Application Objects

Clients cannot obtain access to your Client or Application objects unless the service is in the running state. If your service is not in the running state, attempts to create client objects, or to call the GetAppObject method of the RunningService object will fail with a “Service not active” error.

However, once a client is holding a reference to your Application or Client object, method and property calls to the object will go through as long as the object exists – it is your responsibility to keep track of the current state of the service and handle client requests accordingly.

Once your service stops, any attempts by the client to access methods or properties on an object reference they are holding will result in an “Object Disconnected” error.

It is very important from a design perspective that your client applications are prepared to handle object disconnections any time they access your client object!

Your client object can use OLE callbacks to notify clients when the service is stopping. Your client object cannot raise events to clients, however you can expose additional objects to the client that does raise events.

Use OLE Callbacks, not Events

Because your client object is not actually seen by the client, it cannot raise events to the client. However, you can raise events to the client by creating a new object and passing that to the client, and having that object raise events. Be careful though – you must be sure to free any such sub-objects before you allow your service to stop! If you are confused about the difference between OLE callbacks and events, refer to the book “Developing COM/ActiveX Components with Visual Basic 6” by Dan Appleman.

Security and Impersonation

Few subjects in Windows are as complex and intimidating as security. This section can't possibly cover all of the issues that you may face when configuring DCOM and COM and security, but will at least try to give you a handle on basic security operations relating to services.

NT/2000/XP Security in 250 Words or Less

Every thread in a system belongs to somebody. Each "somebody" is identified with a user name, and may belong to one or more groups. For example: most services run under an account named "LocalSystem" which is a fictional user that exists on every Windows system. Each "somebody" has certain rights or privileges – things that they are allowed to do. For example: the LocalSystem account is allowed to run as a system service (which, among other things, means that it can continue to run even when a user logs out).

There are a variety of securable objects on a system – objects whose access can be allowed or restricted based upon the user. Examples of securable objects include files, registry entries and even COM objects. You can, for example, decide who is allowed to access the Client object that you expose in your service.

Every security operation in Windows consists of answering one of the two following questions:

- Is a particular user allowed to perform a certain operation on a system?
- Is a particular user allowed to perform a certain operation on a securable object?

That is, in a nutshell, everything there is to know about NT/2000/XP security. Everything else is commentary. But oh my, what a commentary it is....

Impersonation

A key concept that you must understand to take full advantage of the service framework is that of impersonation. One of the privileges that the LocalSystem account has is the right to impersonate users. Remember, every thread belongs to a particular user. But there are times where you will want a thread to act on behalf of another user.

For example: Let's say you have a client object exposed from a service. This object has a method that reads a private data file. But you want it to only work for authorized users. How can the object figure out if a particular client is authorized to read the file? You can't just try reading the file – since the object runs in a service thread which is probably always authorized to read the file.

The solution is for the thread to temporarily impersonate the client – effectively pretending to be that client. While impersonating the client, Windows treats the thread as if it were that client, and applies security based on that client. If the program tries to access the file while impersonating, Windows will verify if the client is allowed to access the file – if not, a permission error will occur.

Types of Impersonation

This sounds fairly simple, and really is in concept. In practice, however, it is made more complex by the fact that there are different types of impersonation. Each client machine can decide what level of impersonation it allows. The levels of impersonation are as follows:

- | | |
|-------------|---|
| Anonymous | The client machine does not allow impersonation of its clients. |
| Identity | With this level (the default on many installations), the server can determine who the client is and impersonate the client in order to perform security tests. However, it cannot actually do anything on behalf of the user.

If you wanted to check if a user could access a file using this level of security, you would have to first use API functions to obtain the DACL for the file (a structure that determines which users can access the file), then impersonate the client, then call the AccessCheck API to check whether the operation would succeed. You could not simply try to open the file, because Identity level impersonation does not allow you to do perform the Open operation even if the user has permission to open the file. |
| Impersonate | This is the most useful level of security for impersonation, in which the server is allowed to act as if it were the client in most respects. Instead of the relatively complex operation of checking whether a client can perform an operation, you can attempt try |

the operation. If the client is not allowed to perform the operation, it will fail with a permission error.

There is one small catch to this level of impersonation. You can only access resources on behalf of the user on the local system. You cannot impersonate the client and then use the client's rights or permissions to access resources on remote systems.

Delegate This type of impersonation is only supported by Windows 2000/XP. It is complete impersonation, in which the server can impersonate the client and act on their behalf in every respect – even going to other systems or network based resources.

It is important to remember that the type of impersonation is specified by the client – NOT the server. The client computer decides to what degree the server can impersonate its users. If you wish to use an impersonation level of “Impersonation”, you must set the default level for the client computer to this level using the dcomcnfg application. You must do this even if you are running the client on the same computer as the service!

In addition to impersonating clients, your service can use the LogonUser API (if running under the LocalSystem account) to log in as a particular user and then impersonate that user using API calls (do not use the dwSecurity object to impersonate a user that you logged in using LogonUser – use the Impersonate and RevertToSelf API calls in that case).

Configuring Your Service for COM/DCOM Client Access

Now that you know the basics of security and impersonation, let's take a look at how these concepts work in practice. For the sake of this discussion, the term user and client will be used interchangeably – since every client we will be dealing with also has a user identity.

First, keep in mind the key question behind NT/2000/XP security – does a user have permission to act on an object.

This question divides into two parts: identifying the user, and setting the necessary security so the user can perform the operations in question.

The first part of the question – identifying the user, implies that the computer that the service is running on must somehow be able to identify the user. The question of authentication is complex – depending upon the operating systems in question, how they are configured, how they are organized into domains and forests, and so on. This document will not even begin to address this issue.

Desaware cannot provide technical support on the issue of authentication of users or basic COM/DCOM functionality on your service's computer. Before you call us for technical support on subjects related to client access, you must verify that you can create a standalone Visual Basic EXE on your service's computer, and access it via DCOM from the client system in question, where your client system is logged in under the same account that you wish to use to access the service. If you cannot do this, you have an authentication or connection problem that does not relate in any way to our Toolkit. If you call us for support on what turns out to be an authentication problem, we will charge you our standard consulting rates for time spent on the problem. Authentication and account management issues must be resolved by your system administrator.

The easiest way to be sure that a client account will be authenticated on the service's computer is to be sure that both machines belong to the same domain, and that the client is logged into an account in that domain.

The easiest way to make sure that your computers are configured properly is to create a simple ActiveX EXE server in VB that uses remote automation. Register its VBR file on the client and make sure you can access the server from the client machine. If you do not know how to do this, refer to the Visual Basic documentation and learn how to do this before you proceed to attempt this with a service.

Configuring the Service Access Through DcomCnfg

The following steps are necessary to configure your service to expose objects for remote access via DCOM.

1. The first step on the server side is to compile the service component and register the service to run as a service (by executing it with the parameter `-I`). You cannot remotely access DCOM objects while your service component is running in the VB IDE. (You can, however access objects if the client is on the local system, and thus you should start your testing and debugging using the VB IDE and local clients.)
2. Use `Dcomcnfg` to select the properties for the service application.

Set the launch and access properties to include the client user. If the default access and launch permissions are adequate, you will already see the user on the list when you click the custom access permissions option and select "Edit." If that is the case, you can safely use the default permissions. Default permissions are also set using the `dcomcnfg` program. If you don't see the client user on the list of the computer or available domains, you probably have a user/authentication problem. Contact your system administrator to solve this problem before continuing. Rather than setting security for each possible user, you'll probably actually want to authorize groups to which the potential users belong.

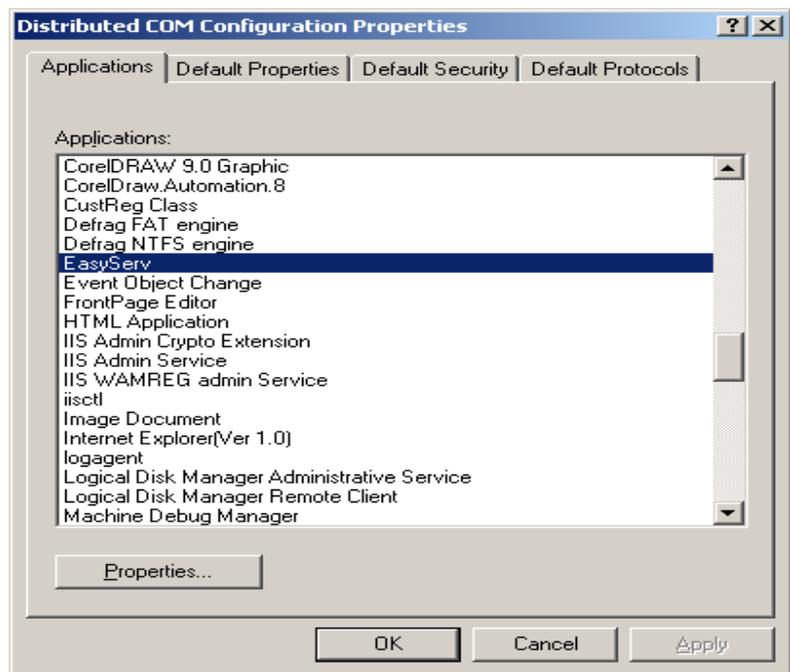


Figure 7
Distributed COM Configuration Properties

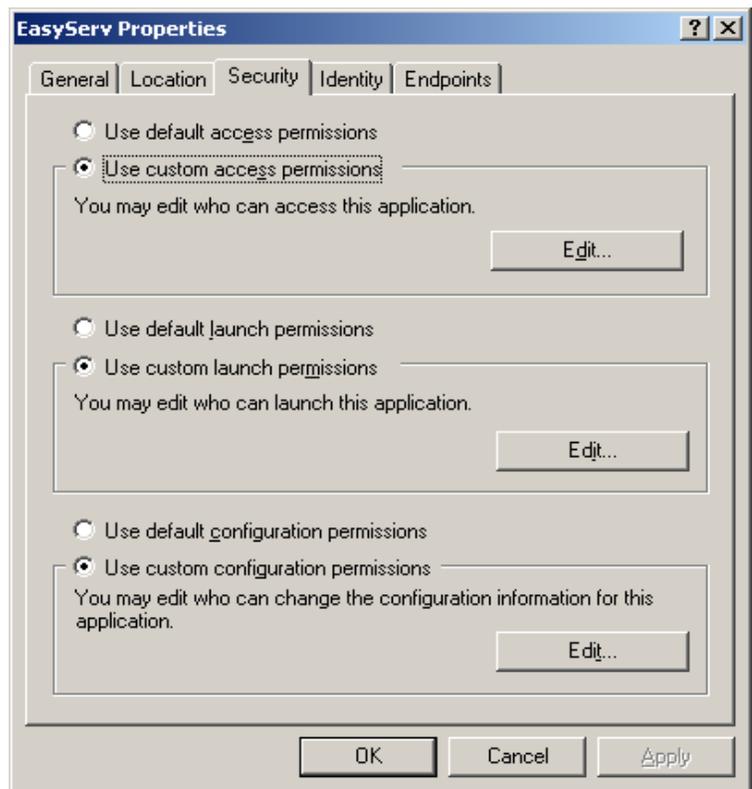


Figure 8
EasyServ Properties

You can use the `dcomcnfg` program to perform other tasks such as setting the log in name and account for the service if you wish to override the settings in the service itself. Note that you must make these changes after the service is installed, and they will be reset if you reinstall the service.

Configuring the Client System for Access to COM/DCOM Objects Exposed from the Service

Before a client can access an object from a service, it must know about that object. It needs three pieces of information:

- The name and class identifier (CLSID) of the object, and other information needed to register the object.
- Information on the interfaces used by the object, and how to invoke them remotely (i.e. marshal the information from one computer to the next).
- The machine on which the object can be found.

The Service Configuration program allows you to create a VBR file. This file contains the information on the objects exposed by your service, along with the application name and identifiers of the service.

The Toolkit also includes the NT Service Type Library, dwNTServ.tlb. This type library contains the interfaces used by the Toolkit and must be registered on the client system in order for it to know how to communicate with the objects in the service. The service toolkit uses type library marshalling, meaning there are no other DLLs to distribute to the client.

Copy the VBR file and dwNTServ.tlb file into the same directory on the client system. Then run the clireg32 program to register the VBR file (the type library will be registered by the VBR file). The CliReg32 program will prompt you for the IP address or name of the machine.

Finally, set the default impersonation level.

- If you aren't using impersonation, you can ignore this setting.
- If you are using impersonation and API techniques to explicitly check security on objects, Identity level impersonation is sufficient.
- If you are actually performing operations on the local system in the security context of the user, set Impersonate level impersonation.
- And if you truly want to *be* the user, and are running Windows 2000/XP, choose delegate level impersonation.

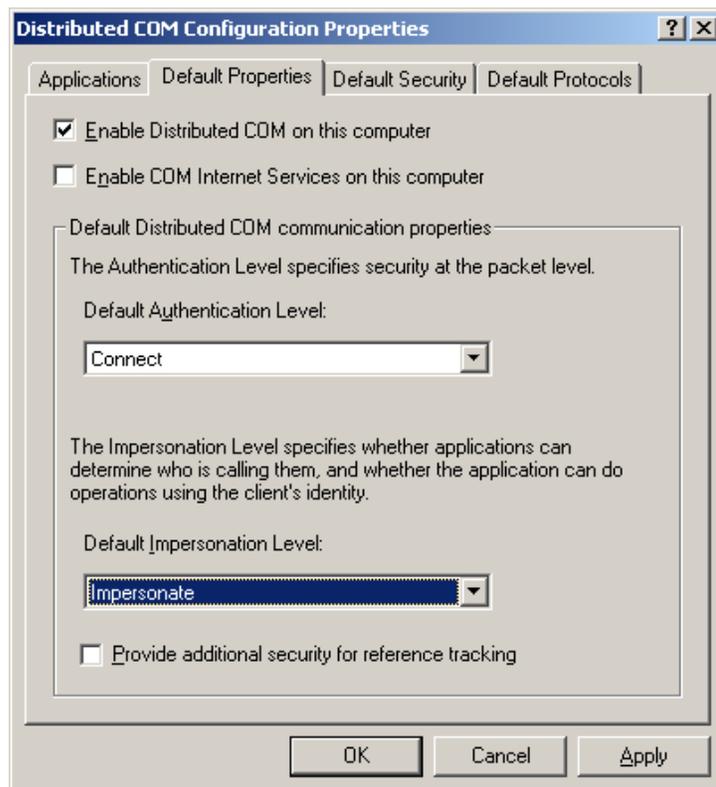


Figure 9
Distributed COM Configuration Properties

Acting on Behalf of Clients

Remember that each thread belongs to one (and only one) user at a time. When you impersonate a user, suddenly you will only have access to those capabilities that the user has. Typically you will only impersonate users briefly for operations that must take place in their security context.

On the other hand, because the default is for client threads and the service thread to run in the service account (typically LocalSystem), it is possible for the service to perform operations on behalf of a client that the client is not allowed to perform directly. Thus services provide an excellent mechanism to expose privileged functionality in a controlled manner to users.

The dwSecurity Object

The Desaware NT Service Toolkit comes with a component named the Desaware SecurityHelper Library (dsvclnt.dll) that exposes an object named dwSecurity that will help simplify a number of security related tasks, especially when exposing client objects through COM.

These methods are exposed through a separate DLL in order to allow them to be used successfully both from within a compiled service, and while debugging using the Visual Basic IDE (the security functions must run in the current process).

***Note:** Do not hold a reference to a dwSecurity object between method calls. The object is intended to be created during a single client method call and freed before it returns.*

Refer to the section on Security and Impersonation for further information on this subject.

The object provides the following methods at this time (check the online documentation for information on methods added since this manual was printed).

Impersonate()

This method starts impersonation of the client on the current thread. The client must allow impersonation (note that the default on most systems is to allow only identification level impersonation, which is not sufficient for this operation to succeed).

Note – if this method is called during a method that was not invoked from a remote client, it will still impersonate the current process (this is known as self-impersonation). This means that if your system does not have the impersonation level set to “Impersonate” you will start seeing security errors if you perform any secured operations!

RevertToSelf()

End impersonation of the client on the current thread. The RevertToSelf() method is automatically called internally when the dwSecurity object is freed.

GetUserInfo

(UserName As String, DomainName As String)

This method is used to obtain the user name and domain of the account that invoked a method on the client object. If the object is not being invoked remotely, it will return the user name and domain of the local client accessing the object.

Creating Background Threads (The dwBackThread Component)

The Desaware NT Service Toolkit includes the dwBackThread component included in SpyWorks Professional. This component allows you to create objects in your service in independent threads and to make asynchronous method calls on those objects. This is important for services that need to scale to support multiple clients, but are supporting the clients through channels other than COM (for example, through Winsock).

The TinyWeb2 example illustrates the use of the dwBackThread component to allow a Winsock server to perform the operations requested by different connections on different threads, thus preventing one connection from locking the service.

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

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

dwBackThread - Quick Start

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

1. Using the Project-References menu, add a reference to the Desaware DLL background thread component.
2. Create a dwObjLaunch object for each background thread you wish to use.
3. Create a class to run in the background thread. The background class should have a subroutine named ExecuteBackground that takes no parameters.
4. Declare a variable to hold a reference to the class (you can declare it With Events if you wish).

5. Create an instance of the background class object using the LaunchObject method of the dwObjLaunch object.
6. Set the properties for the background class object. Then use the dwObjLaunch BackgroundExecute or BackgroundExecuteDelayed methods to launch the background operation asynchronously.

Let's take a closer look.

```

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

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

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

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

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

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

```

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

```

Option Explicit

Event GotExecute(ByVal ThreadID As Long)

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

```

dwBackThread – Methods and Properties

The dwObjLaunch object has the following methods and properties.

LaunchObject(ObjectName As String) As Object

ObjectName is the full name of the object in the form “component.class”. This method creates the background thread and creates the specified object in that thread, returning a proxy to the object that can be referenced from the calling thread. If you have already used this method to launch an object in a thread, calling this method again will release that object and create a new one. Be sure to free references in your program held by that object before using this method to launch a new one!

BackgroundExecute()

This method causes the background object’s “ExecuteBackground” method to be called. An “invalid method or property name” error will be raised if the background object does not have an ExecuteBackground method, or does not support IDispatch (all VB objects support IDispatch). An error will also be raised if a background execution is already in progress. The background object's ExecuteBackground method will be called at some arbitrary time after this method is called depending upon the manner in which the system schedules threads. You cannot assume that the background operation will have already started when this method occurs – in fact, it usually will not have started.

BackgroundExecuteDelayed()

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

BackgroundObject

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

dwBackThread – Summary of Rules

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

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

- Be sure to test your background object as a compiled DLL – it will work, but will not exhibit multi-threading when tested within the VB development environment.
- Each dwObjLaunch object creates a single object on a thread that it manages. You can, however, create as many dwObjLaunch objects as you wish, subject to limitations on the number of threads in the system.
- If you pass object references to your background object, be sure to pass them with ByVal. Marshalling ByRef objects across threads in VB seems iffy (to say the least).

Examples

Your Toolkit includes a number of service examples. These examples are designed to also be part of the documentation – you’ll find the comments include in-depth discussions of the example, its design, and potential issues while testing and debugging the example.

All of the examples for the demo edition have the same name (dwEasyService) and are designed to run with the demonstration version of the framework. For this reason, you can actually only test or run one example at a time.

If you have purchased the Toolkit, all you need to do to use these service samples independently is to use the configuration program to create a service framework executable that references the sample project.

The following examples are included (by subdirectory name).

Template	Contains blank templates for the ServiceConfiguration and Service classes that you can load into a new service project (thus saving some typing).
Beeper	Simple service that beeps at a set interval. Demonstrates conversion from NT service classes in previous versions of SpyWorks.
ControlPanel-Applet	Sample shows how to create a control panel applet that interacts with your service. Also shows how to create a standalone control panel applet.
ReportEvent2	Similar to the Beeper sample. Demonstrates how to use the ReportEvent2 function to perform advanced event logging operations.
Tracing	Demonstrates several of the new features of version 2.0. Similar to the beeper sample, but supports InstallParameter and StartupParameter arguments for different results and to output debugging information.

Launcher	Launches an application, monitors for it to be closed, and relaunches it – making for “unkillable” programs. Demonstrates use of the Application object and background wait threads.
TinyWeb	This primitive web server returns a standard page using the HTTP protocol. Demonstrates non-COM based clients. This example requires SpyWorks or the full version of the Toolkit to run.
TinyWeb2	This slightly less primitive web server returns the same page using the HTTP protocol. Demonstrates use of the dwBackThread object to service HTTP requests on multiple threads, thus preventing connections from blocking each other. This example requires SpyWorks or the full version of the Toolkit to run.
FileWatch	<p>This service watches for changes to files in a directory and reports them via MSMQ to a client application. Demonstrates the use of a background wait thread and MSMQ integration. Classic example of a “monitor” style service, that runs always in the background, queuing information for an application to read later.</p> <p>MSMQ can also be used to allow applications to send messages to a service. This example requires that MSMQ be installed to run.</p>
RemoteUser	<p>This service allows users to access a database record set and a file using COM or DCOM.</p> <ul style="list-style-type: none"> • Demonstrates COM/DCOM access. • Demonstrates pre-loading of data for rapid availability. • Demonstrates performing a high privileged operation on behalf of a low privileged user (acting on behalf of a client).

- Demonstrates holding a service open until all clients disconnect.

Common Errors

Installation and Registration

Service Cannot be Deleted Error When Trying to Install or Delete a Service

Chances are the services control panel applet (NT4) or snap-in (Win2K) is visible and the service is already installed (this typically happens when you wish to reinstall after changing some parameters). Close service window to allow Windows to release its handles to the service, thus allowing the service to delete the current version of the service.

Unable to Load Service Configuration Object Error When Trying to Install a Service

Chances are the corresponding service DLL is not registered, or there may be more than one of them registered (project or binary compatibility was not set for the service DLL).

Debugging with Visual Basic

Permission Denied Runtime Error While Accessing the Service Control Object

Use the Dcomcnfg application to give access permission to the objects in your application to the current logged in user. This is necessary because even though you are running on the same machine, there are two users involved, the LocalSystem user running the service, and the logged in user who is running VB.

Service Stops Suddenly Instead of a Runtime Error Appearing in your VB Code While Debugging

The service framework is designed to stop when a runtime error occurs in your service object. It is up to you to handle all runtime errors. Be sure your error trapping options (in the General tab of the Tools-Options dialog box) are set to "break on all errors", or "break in class module". If you have it set to "break on unhandled errors", the service framework will receive the error and interpret it as a failure in your component and terminate the service.

When logging is enabled, the framework will detect unhandled runtime errors in your component and report them in the log file.

Client Objects

Permission Denied Error When Creating the RunningService Object

Use the Dcomcnfg application to allow your client permission to access the object.

My Client Object is not Receiving an OnStop Method Call

The service object controls termination of the service – thus it is possible for the service to stop before the framework has had time to send the IdwServiceClient_OnStop method on each client object. It is up to your program to defer the service shutdown until the method has been called for all clients if this is important to your particular application. Refer to the description of the IdwServiceClient_OnStop method for details. Refer to the RemoteUser sample application for an illustration of one approach for handling this situation.

A Method Works in VB Debug Mode, but Fails When Compiled

If your client object is accessing a method or property in the service object, be sure the method or property is defined as public. Friend methods and properties will not work correctly.

Unable to Access an Object Through DCOM

This can result from a variety of DCOM configuration errors. The key steps to remember are as follows:

- Use Dcomcnfg to ensure that your service allows both launch and access to the client account.
- Use Dcomcnfg to give the client computer an impersonation level of “Impersonate”.
- Be sure the client account can be authenticated by the server (i.e., both are on the same domain).
- Be sure you are running as a service with a compiled VB component.

- Be sure the VBR file is registered on the client system using the clireg32 application, and the dwNTserv.tlb type library is in the same directory as the VBR file when you do the registration.

While Running

The Service Stops Working

It is critical that your service component not raise unhandled runtime errors. These errors will cause a memory exception and your service will be terminated. Use the tracing and logging capability to detect when this has occurred. Setting a background “beep” on the main service timer provides a handy way to know that a service is running while debugging.

The Service Cannot Access Network Resources When Running as a Service

The LocalSystem account, under which most services run, does not have network credentials, so is not able to access network resources.

This leaves you with two choices as to how to proceed if you must use network resources:

1. Run your service in a user account. You can configure your service to run as a particular user. In this case, be sure you modify the user account so it has permission to run as a service.
2. In Windows 2000/XP, you can use the LogonUser API to log your service onto a specified user account and impersonate that user (with delegation level impersonation) to use their network credentials.

Licensing Issues

There are no royalty fees to distribute services you create with your Toolkit or files listed in the license agreement as redistributable.

The Desaware NT Service Toolkit is licensed to you on a per-computer basis. You must install the framework using the installation program and a unique license key on each computer on which you wish to do the following:

- Any software development using the Toolkit.
- Creation of the service framework executable using the configuration utility.
- Running the service using the simulator.

Upon installation, the license key is associated with the name of the computer on which the software is installed.

If you wish to transfer the license from one computer to another, you must uninstall the software from the first computer before installing it on the second. You may also need to rebuild your service framework executable using the Service Configuration Wizard before you will be able to debug the service on the new machine. Changing your computer name may require reinstallation of the software and rebuilding your service framework executable as well.

Please contact Desaware for information on purchasing site licenses for five or more computers.

Testing and Debugging

Surprisingly, it is in many ways actually easier to debug services written using Visual Basic and the Desaware NT Service Toolkit than it is to debug traditional services written using Visual C++! This is because Visual Basic provides excellent support for the debugging of ActiveX DLL components. In fact, you'll be able to do most of your debugging using the Visual Basic environment.

Tracing and Logging

The Desaware NT Service Toolkit framework is fully equipped to help speed debugging and diagnose problems both during development and after deployment.

You can initiate logging by placing a private initialization file in the same directory as the service executable. The file should have the same base name as the executable with the extension `.ini`. For example: If your service is named `myService.exe`, the initialization file should be named `myService.ini`.

The file should contain a section name "Service" and an entry named "TraceLevel" as follows:

```
[Service]
TraceLevel=4
```

The TraceLevel setting determines what information is written to the log file. When the INI file is missing, or the TraceLevel is set to zero, no log file is created. A TraceLevel of 1 only records severe (fatal) errors. Higher trace levels report additional information that includes additional details on errors, warnings and general information indicating normal operation.

The log file is written to a file with the same name as the service executable and the extension `.LOG`. New entries are appended to an existing file.

When logging is enabled, the framework reports detailed information on errors that occur. Logging works both when the service is running and during installation and uninstallation. The framework will also trap and report any unhandled runtime errors that occur in your service component.

You may use the Trace method of the `IdwServiceCtl` interface to write information to the log file from your service component.

Testing and Debugging – Simulator Mode

The majority of your initial debugging can be done using the VB environment in the service framework's simulation mode. To perform this type of debugging, do the following:

- Open your Visual Basic component in Visual Basic.
- Run your project (specify to “wait until the component is created” in the project Debug options).
- Register your service executable with the command line parameter “-RegServer” (or drag and drop your service executable file on the “Register for Simulator” command button in the service launcher utility).
- Run your service executable with the command option “-Sim” (or drag and drop your service executable file on the “Run Service in Simulator mode” command button in the service launcher utility).

The simulator window will appear that allows you to Pause, Continue, Stop and Shutdown your service. You can set breakpoints in your Visual Basic component and verify the operation of the component using the Visual Basic environment.

***Note #1:** Be sure your debug options are set to “Break on all errors” or “Break on class errors”. If you have it set to “break on unhandled errors”, the service framework will receive the error and interpret it as a failure in your component and terminate the service.*

***Note #2:** If you have a background timer set, the timer method will continue to be called while in break mode. This can make stepping through a program a bit tricky – the IDE will keep flickering and it may be difficult to edit text in the module or immediate window. Add some debug code to stop the timer at your breakpoint to avoid this problem if you run into it. This applies while debugging using VB and running as a service as well*

Testing and Debugging – While Running as a Service

While the simulator does a fairly good job of mimicking the behavior of a service, it does not do a complete job because in simulator mode the service executable runs as a standalone application in the security context of the logged in user. You must verify operation of your service while running as a service. Even so, you can continue to debug the service using Visual Basic by doing the following:

- Open your Visual Basic component in Visual Basic.
- Run your project (specify to “wait until the component is created” in the project Debug options).
- Register your service executable with the command line parameter `-i` (or drag and drop your service executable file on the “Install Service” command button in the service launcher utility).
- Run your service executable using the system administrative tools. Please refer to the following **Note** for instructions as to how to proceed should you receive a “Permission Denied” error.

You can use the Service administrative tools to start, stop, pause and continue your service. You can set breakpoints in your Visual Basic component and verify the operation of the component using the Visual Basic environment.

Note: If you receive a “Permission Denied” error while accessing objects exposed by the service framework, chances are you need to configure the security settings for the application using the DcomCnfg program to allow access to your application from the current logged in account. Refer to the section on Security and Impersonation for further details.

Testing and Debugging – Full Native Mode

There are several service features that cannot be debugged using the Visual Basic environment. This is not a limitation of Visual Basic or the framework, but rather a consequence of the fact that when you use the Visual Basic debugger, the service component is in a different process, and runs in a single thread. This has the following implications:

- You can't detect synchronization problems when testing in the Visual Basic IDE, since all objects (including the client object) run in the primary thread of the development environment.
- You can't catch termination problems when testing in the Visual Basic IDE for the same reason.
- While running in the Visual Basic IDE, your component is running in the security context of the logged in user (who launched Visual Basic) and owns a desktop. If your service uses features that depend on either of these facts, it will probably fail when running as a true DLL.
- Depending on the security configuration of your system, you may not be able to test client objects accessed via DCOM when testing in the Visual Basic IDE.
- API objects (such as synchronization objects) that can typically be shared between an application and DLLs, cannot be shared when the DLL is being tested out of process. The service framework handles this case for synchronization objects that you with the SetWaitOperation method of the ServiceControl object. But there may be other situations where this becomes an issue (no, we don't know of any practical illustrations of this at this time, but the theoretical problem exists).

It is essential that you test your service thoroughly as a fully compiled service running as a service.

To debug your service as a service, you'll need to use the Visual C++ or other Windows debugger. The steps to debug a service using Visual C++ are as follows:

- Compile your Visual Basic component. Set the compilation parameters to "no optimization", and "Create Symbolic Debug Info".
- Register your service executable with the command line parameter `-i`.
- Run your service executable using the system administrative tools.

- In Visual Studio, select the “Build – Start Debug – Attach To Process” menu command (see below for what to do if this list is empty).
- Select “View system processes”.
- Start debugging on the process for your service.
- You can set breakpoints by opening a VB source file for your component and setting a breakpoint on the desired line.

If the list of processes is empty, you are probably dealing with a known bug (see MSDN knowledge base article Q235434). To solve this problem, use the Tool-Options menu command (Debug tab) to enable “Just-in-time debugging”. To debug a process, use the task manager, right click on a service process, and select Debug.

Tip: Use the *MessageBeep* API in the *OnTimer* event in the service, with a timeout to give you an audible indication that your service is running.

Testing COM and DCOM

Our experience suggests that the best strategy for testing services that expose objects through COM and DCOM is to first test them using the simulator and Visual Basic with the client running on the same system as the service (note that you will still need to set security using *Dcomcnfg* and set the impersonation level to “Impersonate” for this to work).

When you are ready to test remote access, you should go directly to running as a service using a compiled component. Each of the possible combinations (simulator vs. service, component in VB vs. compiled) has its own set of security issues, and solving the problems in one will not necessary resolve those for others. You will not be able to access objects remotely when running your VB component in the VB IDE.

dwSCM Component: Service Control Manager

The Service Control Manager (SCM) is a series of Windows API functions that allow you to control, get information about, and configure services. The dwSCM component gives you this functionality in Visual Basic.

dwSCM Architecture

The dwServiceManager is the base object. It represents the Service Control Manager itself. It allows you to access services, both running and inactive. You can get information about a service with as little as one of the service's strings. This object also allows you to register service executable files.

The dwServiceObject object represents a single service. This controls all the command and configuration settings for a single service. It can also delete a particular service from the list of services.

The dwServiceConfig object contains the configuration settings for a service, such as the dependencies or the account the service uses. This is used with the configuration functions in dwServiceObject.

The dwServiceStatus object contains the description of a service and its current status. It is returned by the QueryServiceStatus method of dwServiceObject.

When errors take place, dwSCM will raise an error using Windows API error values with descriptions for common errors. When first using the component, you will become very familiar with error number 5, which means you do not have the proper permission to perform a certain task. Opening a service without a needed flag usually causes this error.

dwServiceManager Methods

InitializeSCManager

(ByVal Machine As String, ByVal Database As String, ByVal Access As ServiceControlRights) As Long

Use this method to initialize the Service Control Manager, which allows you to use the other methods of this object. The Machine parameter allows you to access other machines on the network. Leave this blank to specify the local machine. When specifying a foreign machine, be sure to prefix the machine name with “\\”. The Database parameter specifies which service database you would like to manage. Normally this would be blank. The Access parameter describes what types of functions you will be using. You would almost always use the SC_MANAGER_ALL_ACCESS flag, which permits use of all the functions. See MSDN documentation for the OpenSCManager API function for information on the other, more limited flags. This function returns the handle to the manager. This can be used if you need to call your own API functions with such a handle. It is not needed for any functions in the dwSCM library.

```
Dim sc as New dwServiceManager
Call Sc.OpenSCManager ("", "", SC_MANAGER_ALL_ACCESS)
```

EnumServicesStatus

(Optional ByVal ServiceType As Long = SERVICE_WIN32, Optional ByVal ServiceState As EnumServiceStates = SERVICE_STATE_ALL) As Collection

This method returns a collection of dwServiceStatus objects, each one containing information on the current status of a service. ServiceType in an optional parameter that describes which kinds of services are enumerated – either driver service (SERVICE_DRIVER), normal services (SERVICE_WIN32) or both (SERVICE_DRIVER And SERVICE_WIN32). ServiceState is an optional parameter of type EnumServiceStates that specifies if running, non-active, or all services should be enumerated.

```
Dim svcstat As dwServiceStatus
Dim ServiceCollection as Collection
```

```

' This will add a dwServiceStatus object to the
' ServiceCollection for each service. Use the
' default parameters to get a list of non-driver
' services, both running and not.
Set ServiceCollection = sc.EnumServicesStatus()
' Now the names and descriptions are printed.
For Each svcstat In ServiceCollection
    Debug.Print svcstat.Name
    Debug.Print svcstat.DisplayName
Next

```

OpenService

(ServiceName As String, DisplayName As String, ByVal Rights As ServiceAccessRights) As dwServiceObject

The OpenService opens the specified service, allowing you to control it and to access its configuration. ServiceName is a string which contains the service name. DisplayName is a string which contains the user-friendly display name. These strings can be obtained by using the EnumServicesStatus method. Rights (parameter) is a combination of the values in the ServiceAccessRights enumeration. This describes what actions you plan to take with the service. This method returns a dwServiceObject object.

```

Dim svcstat As dwServiceStatus
Dim OpenService as dwServiceObject
' Open the service to change its configuration.
Set OpenService = sc.OpenService (svcstat, _
    SERVICE_CHANGE_CONFIG)

```

GetDisplayNameFromServiceName

(ByVal ServiceName As String) As String

GetServiceNameFromDisplayName

(ByVal DisplayName As String) As String

Each service has two names: a service name that is its key in the Registry, and a user-friendly display name that is displayed in the Service control panel applet. You can use these functions to learn one name by using the other.

CreateService



**ByVal ServiceName As String, ByVal
DisplayName As String, ByVal DesiredAccess As
ServiceAccessRights, ServiceConfig As
dwServiceConfig) As dwServiceObject**

If you have a service executable on your machine, you can add it to the list of services by using this method. This also allows you to set all of the configuration information. It will then open the service and return a `dwServiceObject`. `ServiceName` is the name the service identification. It must be less than 256 characters and not contain slash or backslash characters. `DisplayName` is the user-friendly description of the service's purpose. It should also be less than 256 characters. Parameter `DesiredAccess` is a combination of the flags in the `ServiceAccessRights` enumeration. It represents what actions you need to perform on the service. `ServiceConfig` is an `dwServiceConfig` object that represents the initial configuration of the service.

You should not use this method for registering a service created using the Desaware NT Service Toolkit. Such services already have a built in ability to register themselves, and take their configuration from internal settings.

LockServiceDatabase () As Boolean

It is possible for services to start while you are trying to change the configuration. This will not be a problem if you are only performing one action, but if you are performing multiple actions and the service starts during your operation, it could cause problems. To prevent this, you can use the `LockServiceDatabase` method, which prevents services from starting. `TRUE` is returned if the SCM was successfully locked.

UnlockLockServiceDatabase () As Boolean

If you have used `LockServiceDatabase` method, be sure to call this immediately after you have completed modifying a service.

QueryLockStatus (LockOwner As String, Duration As Long) As Boolean

If you cannot start a service, this can assist you in determining if it is because another program has locked the Service Control Manager. It will fill the LockOwner and Duration parameters with the corresponding information about who created a lock. FALSE is returned if the SCM is not currently locked.

```
Dim retval as Boolean
Dim LockOwner as String
Dim Duration as Long

retval = sc.QueryLockStatus (LockOwner, Duration)
If (retval = True) Then
    Debug.Print "Locked by: "; LockOwner
    Debug.Print "For "; Duration; "seconds"
End If
```

dwServiceObject Methods and Properties

StartService (Optional ByVal Arguments As String = "")

This will send a request to the service represented by this dwServiceObject to start. If the service uses any command line parameters, you can pass them in the optional Arguments string parameter. Place spaces between each parameter. This function will end before the service has actually started – use QueryServiceStatus to tell when the service has responded.

```
Dim svcstat As dwServiceStatus
Dim OpenService as dwServiceObject

Set OpenService = sc.OpenService (svcstat, _
SERVICE_START)
OpenService.StartService
```

ControlService (Operation As ServiceControlConstants)

To perform other actions, such as stopping, pausing or unpausing the service, this method is used. The Operation parameter is a member of the ServiceControlConstant enumeration. This function will end before the service has actually performed the action requested – use QueryServiceStatus to tell when the service has responded.

```
Dim svcstat As dwServiceStatus
Dim OpenService as dwServiceObject

Set OpenService = sc.OpenService (svcstat, _
SERVICE_STOP)
OpenService.ControlService SERVICE_CONTROL_STOP
```

QueryServiceStatus() As dwServiceStatus

This returns a dwServiceStatus object which describes the current running status of this service.

```
Dim OpenService As dwServiceObject
Dim ss As dwServiceStatus

Set OpenService = OpenSelectedService _
(SERVICE_QUERY_STATUS)
Set ss = OpenService.QueryServiceStatus()
Debug.print ss.DisplayName
```

QueryServiceConfig() As dwServiceConfig

This returns a dwServiceConfig object that describes the service configuration. Note that this does not reveal the password of the account that the service is using.

```
Dim OpenService As dwServiceObject
Dim sc As dwServiceConfig

Set OpenService = OpenSelectedService _
(SERVICE_QUERY_CONFIG)
Set sc = OpenService.QueryServiceConfig()
Debug.print sc.AccountName
```

ChangeServiceConfig (ServiceConfig As dwServiceConfig)

ChangeServiceConfig sets the service configuration according to the ServiceConfig (is an instance of dwServiceConfig) parameter. If the service is currently running, several of the configuration settings will not take effect until the service has stopped.

```
Dim OpenService As dwServiceObject
Dim sc As New dwServiceConfig

' The only thing being changed is the display name.
sc.StartType = SERVICE_NO_CHANGE
sc.ServiceType = SERVICE_NO_CHANGE
sc.ErrorControl = SERVICE_NO_CHANGE
sc.StartType = SERVICE_NO_CHANGE
sc.DisplayName = "New Display Text"
Set OpenService = OpenSelectedService _
(SERVICE_CHANGE_CONFIG)
OpenService.ChangeServiceConfig(sc)
```

EnumDependentServices (ByVal ServiceState As Long) As Collection

If you need to stop a service, you should also stop all services that depend on it – otherwise the dependent services might fail or even cause exceptions. EnumDependentServices returns a collection of dwServiceStatus objects representing all the services that require this service to start first before they themselves can run.

The collection is in reverse order of the start order, with group order taken into account, so you can stop the services in the correct order. The ServiceState parameter lets you limit the list to only those services currently running or paused (SERVICE_ACTIVE), those services currently stopped (SERVICE_INACTIVE), or all dependent services (SERVICE_STATE_ALL).

DeleteService()

This method will delete the service from the list of services. This method might end before the actual deletion takes place. If the service is currently running or has any open handles (such as any dwServiceObjects that reference it) then the Service Control Manager will wait until the service has stopped and there are no more handles open. It is possible that this will not take place until the machine is shutdown and restarted. This does not effect the service files.

ServiceName as String

This is a read only property containing the service name string.

ServiceHandle as Long

This is a read only property that contains the service handle. It can be useful in certain API function calls.

dwServiceStatus Properties

DisplayName as String

Read only property containing descriptive string meant for display.

ServiceName as String

Read only property containing the service name string.

CurrentState as ServiceStateConstants

Read only property describing the current running state of the service. It is a value of the ServiceStateConstants enumeration.

ControlsAccepted as ControlsAcceptedFlags

Read only property describing to which commands the service will respond. See the ControlsAcceptedFlags enumeration below for more details.

Win32ExitCode as Long

This read only property will contain an error value if there is a problem during service starting or stopping.

ServiceSpecificExitCode as Long

This read only property contains a server-specific error value if there is a problem during service starting or stopping, and the Win32ExitCode property is set to ERROR_SERVICE_SPECIFIC_ERROR.

Checkpoint as Long

Read only property containing a value that the service increments periodically to report its progress during a lengthy start, stop, pause, or continue operation. This will be zero when the service does not have an operation pending.

WaitHint as Long

An estimate of the amount of time, in milliseconds, that the service expects a pending start, stop, pause, or continue operation to take before the service either changes the CheckPoint or the CurrentState properties. If the amount of time specified by WaitHint passes, and neither CheckPoint nor CurrentState has changed, you can assume that an error has occurred and the service should be stopped.

dwServiceConfig Properties

ServiceType as ServiceTypes

ServiceType is a value of the ServiceTypes enumeration. It describes the nature of the service executable, such as if it is a driver, or if it runs in its own process or not. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, set it equal to SERVICE_NO_CHANGE.

StartType as ServiceStartTypes

StartType is a value of the ServiceStartTypes enumeration. It describes when the service starts, such as at system boot or at user command. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, set it equal to SERVICE_NO_CHANGE.

ErrorControl as ServiceErrorControlType

ErrorControl is a value of the ServiceErrorControlType enumeration. If the service fails to start, this will determine how seriously the system will take it, and how it will respond. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, set it equal to SERVICE_NO_CHANGE.

BinaryPathName as String

BinaryPathName is string containing the location of the service executable. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, do not modify this property.

LoadOrderGroup as String

LoadOrderGroup is a string containing the name of the group to which this service belongs. Services in a group are loaded together. This can be blank. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, do not modify this property.

TagId As Long

TagID is a long integer which identifies this service within a group (if the service belongs to one).

Dependencies as String

Dependencies is a list of all names of services that this service depends upon to run. Each name is separated by a semicolon.

If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, do not modify this property.

AccountName as String

AccountName is the name of the system account the service logs in under.

If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, do not modify this property.

Password as String

Password is the password for the account specified under the AccountName property. Note that this property is never set by any of the dwSCM functions. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, do not modify this property at all.

DisplayName as String

DisplayName is a user-friendly description of the service's purpose. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, do not modify this property at all.

Description as String

Description is a string that describes the purpose of the service in more detail. It is only valid when the library is being used on Windows 2000 or XP machines. If you are creating a dwServiceConfig instance for use with ChangeServiceConfig and you do not want to change this value, do not modify this property at all.

Enumerations and Constants

ServiceTypes Enumeration

Using the OR command, combine together as many of the following as necessary to describe your service type.

SERVICE_KERNEL_DRIVER	File system driver service.
SERVICE_FILE_SYSTEM_DRIVER	Driver service.
SERVICE_WIN32_OWN_PROCESS	Service that runs in its own process.
SERVICE_WIN32_SHARE_PROCESS	Service that shares a process with other services.

ServiceStartTypes Enumeration

Use the one constant that describes when the service should start.

SERVICE_BOOT_START	A device driver started by the system loader. Valid only for driver services.
SERVICE_SYSTEM_START	A device driver started by the IoInitSystem function. Valid only for driver services.
SERVICE_AUTO_START	Service started automatically during system startup.
SERVICE_DEMAND_START	Service starts when a process calls the StartService function (such as from the Control Panel).

SERVICE_DISABLED	Service that cannot be started. Attempts to start the service result in the error code 1058 (ERROR_SERVICE_DISABLED).
------------------	---

ServiceErrorControlTypes Enumeration

Use the one constant that describes how the system should behave when the service encounters an error during system boot.

SERVICE_ERROR_IGNORE	The error is logged but the startup operation continues.
SERVICE_ERROR_NORMAL	The error is logged and a message box is displayed but the startup operation continues.
SERVICE_ERROR_SEVERE	The error is logged. If the last working configuration is being started, the startup operation continues. Otherwise, the system is restarted with the last known working configuration.
SERVICE_ERROR_CRITICAL	The error is logged, if possible. If the last working configuration is being started, the startup operation fails. Otherwise, the system is restarted with the last known working configuration.

ServiceControlRights Enumeration

Using the OR command, combine together as many of the following as necessary to describe the access you need when you open the Service Control Manager. Note that SC_MANAGER_ALL_ACCESS is a combination of all of the other values, and will give you access to all the methods of the dwServiceManager class.

SC_MANAGER_CONNECT	Required to access the service control manager. It is automatically added by the dwSCM library.
SC_MANAGER_CREATE_SERVICE	Enables the CreateService function.
SC_MANAGER_ENUMERATE_SERVICE	Enables calling of the EnumServicesStatus function.
SC_MANAGER_LOCK	Enables the LockServiceDatabase and UnlockServiceDatabase functions.
SC_MANAGER_QUERY_LOCK_STATUS	Enables the QueryServiceLockStatus function.
SC_MANAGER_ALL_ACCESS	Combination of all the above.

ServiceAccessRights Enumeration

Using the OR command, combine together as many of the following as necessary to describe the access you need when you open a service. Note that SERVICE_ALL_ACCESS is a combination of all of the other values, and will give you access to all the methods of the dwServiceObject class.

SERVICE_QUERY_CONFIG	Enables QueryServiceConfig.
SERVICE_CHANGE_CONFIG	Enables ChangeServiceConfig.
SERVICE_QUERY_STATUS	Enables QueryServiceStatus.
SERVICE_ENUMERATE_DEPENDENTS	Enables the EnumDependentServices function.
SERVICE_START	Enables StartService.
SERVICE_STOP	Enables calling the ControlService function to stop the service.

SERVICE_PAUSE_CONTINUE	Enables the ControlService function to pause or continue the service.
SERVICE_INTERROGATE	Enables the ControlService function to ask the service to report its status immediately.
SERVICE_USER_DEFINED_CONTROL	Enables calling of the ControlService function to specify a user defined control code.
SERVICE_ALL_ACCESS	Combination of all of the above.

ServiceControlConstants Enumeration

Use the appropriate value to indicate which action you wish the service to take.

SERVICE_CONTROL_STOP	Ask the service to stop.
SERVICE_CONTROL_PAUSE	Ask the service to remain active, but not perform any actions.
SERVICE_CONTROL_CONTINUE	Ask a service which is paused to return to normal running state.
SERVICE_CONTROL_INTERROGATE	Ask the service to report its current status information to the service control manager.
SERVICE_CONTROL_SHUTDOWN	Inform the service that the system is about to shut down.
SERVICE_CONTROL_PARAMCHANGE	Inform the service that its startup parameters have changed.

Values between 128 and 256 may also be passed if the service defines the action associated with that control value. The service must have been opened with SERVICE_USER_DEFINED_CONTROL access.

ServiceStateConstants Enumerations

You will receive one of these values to indicate the current state of the service.

SERVICE_STOPPED	1
SERVICE_START_PENDING	2
SERVICE_STOP_PENDING	3
SERVICE_RUNNING	4
SERVICE_CONTINUE_PENDING	5
SERVICE_PAUSE_PENDING	6
SERVICE_PAUSED	7

EnumServiceStates Enumeration

Use the appropriate value to indicate which services you want enumerated when calling the EnumDependentServices method of the dwServiceObject class.

SERVICE_ACTIVE – List services that are running or paused.

SERVICE_INACTIVE – List services that are not active.

SERVICE_STATE_ALL – List all services.

ControlsAcceptedFlags Enumeration

You will receive a value comprised of the following enumeration values that represent the type of commands the service can receive. To find out which flags have been set, AND one of the constants below with the ControlsAccepted property of the dwServiceStatus class. For example:

```
Dim ss as dwServiceStatus
If (ss.ControlsAccepted And SERVICE_ACCEPT_STOP) _
Then Debug.Print "Service accepts stop requests"
```

SERVICE_ACCEPT_STOP If this value is set, the service will respond to stop requests.

SERVICE_ACCEPT_PAUSE_CONTINUE	If this value is set, the service will respond to pause and continue requests.
SERVICE_ACCEPT_SHUTDOWN	If this value is set, the service will be notified of when the system is shutting down.
SERVICE_ACCEPT_PARAMCHANGE	If this value is set, the service can accept new startup parameters while it is running. This is only valid in Windows 2000/XP.

The dwSock Component: Winsock Programming

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

The dwSock component included with this package is part of Desaware's SpyWorks Professional, and like most SpyWorks components, offers a somewhat different approach.

The dwSock6.dll component has several unique characteristics:

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

The current edition of SpyWorks Professional (version 6.2) includes, the dwSock component that is designed for version 1.1 of Winsock. Refer to the online help for a detailed list of methods and properties for the Winsock components.

How to Approach the Winsock Package

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

What does this mean?

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

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

Important Note Regarding Support For This Component

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

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

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

Learning Winsock

A complete description of sockets and the Winsock API is far beyond the scope of this document. You will need to be familiar with the fundamentals of Internet programming in order to take full advantage of this component. The Winsock API documentation in the Win32 SDK or MSDN CD-ROM library will prove useful.

The book "Visual Basic 4.0 Internet Programming" by Carl Franklin (ISBN 0-471-13420-1) or "Visual Basic 6.0 Internet Programming" by Carl Franklin (ISBN 0-471-31498-6) contains an excellent introduction to Internet programming.

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

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

IP Addressing

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

When you contact a system, such as `www.desaware.com`, the name "`www.desaware.com`" identifies a system, but is not the actual IP address. This kind of identifier is also called a URL or Universal Resource Locator (for reasons that are probably well known to Internet mavens but are totally irrelevant for this discussion). When you request this site, your system uses the name resolution capabilities of the Winsock API to locate the IP address assigned to that name. This determination might actually involve a search of many systems over the Internet - but that takes place behind the scenes. In this case the Winsock functions would tell you that the IP address of `www.desaware.com` is `206.169.23.2`. These numbers represent four bytes of data which form a 32 bit IP address.

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

Ports

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

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

Every connection thus has the following attributes:

- A source IP address (your system's address).

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

All four of these attributes form a unique connection. A connection is made up of two sockets, one on each system. The socket is defined by an IP address and port combination.

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

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

UDP and TCP

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

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

dwSock Architecture

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

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

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

Desaware's Winsock component is designed to take full advantage of the asynchronous features of the Winsock DLL. This means that most requests simply start a background operation. This is the only way to provide reasonable performance with multiple connections - otherwise a data request could block all other sockets in use by the application.

The dwSockets object raises events when background operations finish, passing the appropriate object to the client as an event parameter. All of the work of keeping track of which operation refers to which socket, is handled by the component.

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

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

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

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

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

Refer to the online help for a detailed list of methods and properties for the Winsock components.

Refer to the Winsock specification for a complete list of Winsock errors. The constants for these errors start with the prefix WSA and can be found in the GlobalInfo standard module.

The dwSock6.dll object can also raise the following errors:

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

Dependencies

The dwSock6.dll component requires that you distribute the following additional components:

dwspyvb6.dll	The Desaware Windows Utility component.
dwspy5.dll	A utility DLL used by dwspyvb6.dll.
SockIntf.dll	A DLL that is required as an interface to Winsock in order to retrieve accurate error status information.

Using the Winsock Package

The following examples demonstrate the Winsock Utility functions. Additional examples are included with this package including a simple FTP client and a demonstration of direct use of sockets. Refer to the online help for a detailed list of methods and properties for the Winsock components.

Winsock Utility Functions

Obtaining Winsock Version Information

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

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

Obtaining Your Host Name

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

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

Determine the Standard Port Number of a Service

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

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

Perform an Asynchronous Name Resolution

This example from the `SocketTst2` application demonstrates how names can be resolved.

At the module level define:

```
Dim WithEvents skt As dwsockets
```

Start the operation thus:

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

One of the following two events will be raised:

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

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

HTTP Example

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

At the module level define:

```
Dim WithEvents http As dwHTTP10
```

When the form loads, initialize the dwHTTP10 object

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

Here's the command that actually does the request

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

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

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

Creating Control Panel Applets

It is not uncommon to use control panel applets to control or configure services. The NT Service Toolkit includes a framework for authoring control panel applets that is similar to the one used to create NT services. As with services, it allows you to test and debug your control panel applets using the Visual Basic IDE.

Building a Control Panel Applet

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

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

Using the Control Panel Applet Wizard Program

The Desaware Control Panel Applet Wizard program creates a custom control panel applet specifically for your Visual Basic project. This Wizard can also be used to review and edit the information in your compiled Control Panel Applet. This Wizard guides you through a series of steps requesting information regarding your Visual Basic control panel applet. The following describes each step.

Control Panel Applet Name

This is the first step in the Wizard. Enter the name of the control panel applet file. It can be any name you choose. If you select an existing control panel applet file, this Wizard will extract the control panel applet information from that file and initialize the remaining steps of the Wizard with that information. This Wizard can only input control panel applet files created by this Wizard. You can use the Browse button to navigate your file system to specify a file name for your control panel applet. This Wizard will only generate files with the CPL extension.

Tip: You can create a control panel applet file that contains some default information (such as Version Information) for your company or product to serve as a template file. Each time you need to create a new control panel applet, select the template file to initialize this Wizard with the Version Information, etc., then change the Applet file name to the name you want to give for the particular control panel applet.

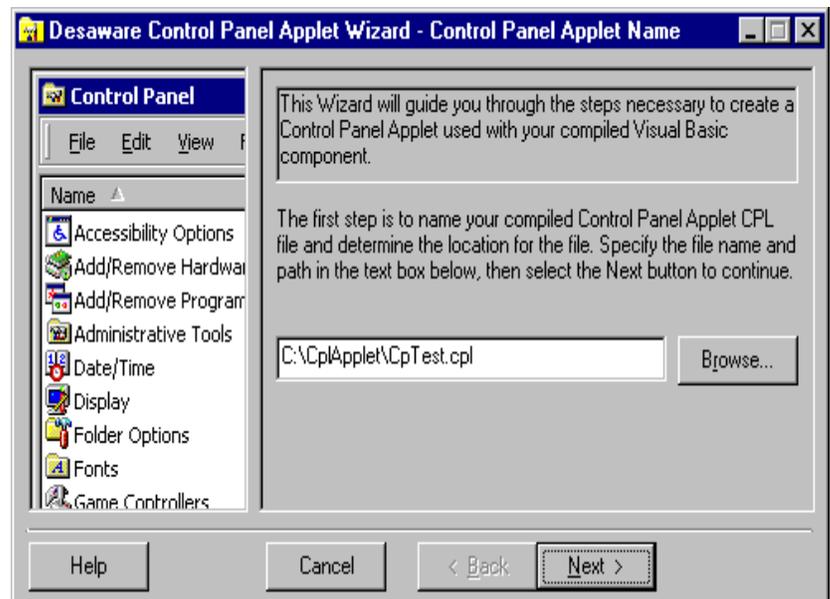


Figure 10
Control Panel Applet Wizard

Project Name

Enter the project name of the Visual Basic DLL that contains the Applet object with which your control panel applet will communicate. If you decide later that you will want to change the project name, you will have to run this configuration program again.

It is important that you should choose a component name that is unique – duplication of project names used by this framework might cause control panel applets to fail to work properly. We recommend including your company name or initials in the name. For example: most Desaware components include the prefix “dw”. The project name length can be up to 32 characters (due to the 39 character limitation for the combined project and class name, and the requirement that the “Applet” class name needs to be exposed). But, your actual project name may need to be shorter depending upon the length of the names of the objects that your project exposes.

Version Information

Enter the version resource information for your service executable file. The control panel applet Wizard writes the same version resource information to your service executable as Visual Basic. You must enter information in the “Company Name” and “File Version” fields. The “File Version” field must contain a valid version number in “#.#.#.#” format, for example “1.0.0.1”. If you select an existing control panel applet file to compile, its file version will automatically be incremented by 1 revision where revision is the fourth version number field in the version format.

Description

Enter your control panel applet’s name and description. The control panel applet name is the string that is displayed in the Control Panel directly below your control panel applet’s icon. The control panel applet description is the string that is displayed next to your control panel applet’s name when the Detail View is selected in the Control Panel. Your control panel applet must have a name and a description otherwise it will fail. If you do not specify a control panel applet name and description in this Wizard, then you must specify the name and description in your Applet class. For more information, refer to the Applet.cls template class file.

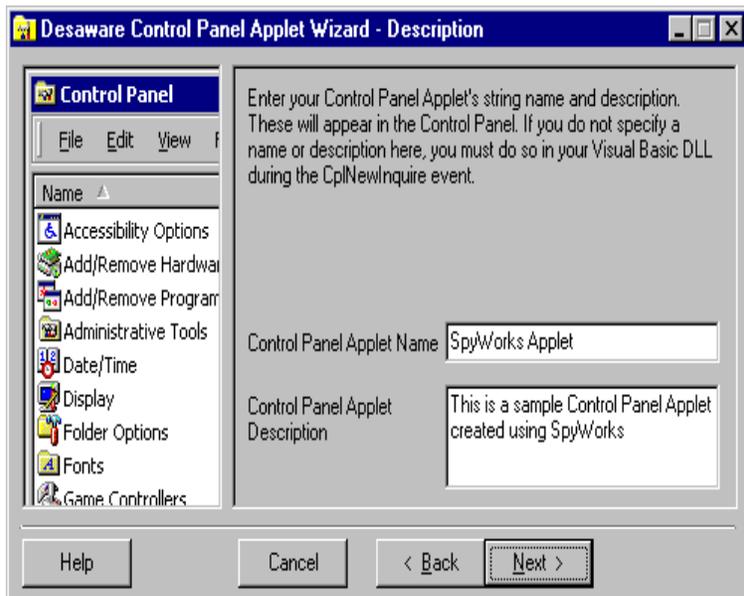


Figure 11
Control Panel Applet Wizard – Description Dialog Box

Icon File

Enter your control panel applet's icon file name. The selected icon file is compiled into your control panel applet and used for display in the Control Panel. If you are compiling an existing control panel applet file, you may choose to select the existing icon from the current control panel applet file instead of selecting an icon file. Your control panel applet must have an icon otherwise it will fail. If you do not specify a control panel applet icon name in this Wizard, then you must specify the icon in your Applet class. For more information, refer to the Applet.cls template class file.

Compile Applet

The control panel applet Configuration Wizard is now ready to compile your control panel applet file. Select the Next button to start compilation, or the Back button to make any changes.

Compile Completed

The Control Panel Applet Wizard has finished compiling your service executable. This step displays whether the compilation was successful or not. After a successful compilation, you should install your control panel applet in the appropriate directory before running it. If you are running systems prior to Windows 2000/XP, you should install the control panel applet file into the System directory. If you are running Windows 2000/XP, you should install the control panel applet file into your application directory and insert the appropriate entry into the Windows registry. Be sure that your Visual Basic project containing the Applet object is already registered prior to running your control panel applet (or opening your Control Panel).

System Compatibility

The Control Panel Applet Wizard requires Windows NT or 2000/XP to run. However, applet programs created by the wizard are compatible with Windows 95/98.

Create an ActiveX DLL for Your Control Panel Applet

Create an ActiveX DLL and create a class named Applet that implements the IdwControlPanelApplet interface as shown here:

```
' Control panel applet example  
' Copyright ©2000 by Desaware Inc. All Rights Reserved
```

```
Option Explicit
```

```
Implements IdwControlPanelApplet
```

The cpapplet.tlb type library contains the definition for this interface, and was registered when you installed the software package. The methods of this interface are called by the applet framework during the course of the applet operation.

CpIDbIClk

(ByVal AppNumber As Long, ByVal UserData As Long)

This method is called when the user double clicks on the applet icon or name in the control panel window. You should display the main form of your applet at this time as follows:

```
frmApplet.Show vbModal
```

If your applet DLL supports more than one applet, the AppNumber parameter will indicate which applet was invoked (from zero through the number of applets - 1). The UserData parameter will contain the same value specified in the CplInquire or CplNewInquire methods.

CplExit()

This method is called before the applet DLL is unloaded. You should perform any final cleanup operations here.

CplGetCount() As Long

Return the number of applets in your applet file as the return value for this method. A single applet DLL can support multiple applets, but this is very uncommon. You will typically just include the following line in this method:

```
IdwControlPanelApplet_CplGetCount = 1
```

The AppNumber parameter for the other IdwControlPanelApplet methods will range from zero to one less than this number.

CplInit() As Long

This method is called when your applet is loaded. You should return the value 1 as a result if your initialization succeeds. If you do not return one, or return zero, the applet will fail to load.

```
IdwControlPanelApplet_CplInit = 1
```

CplInquire

(ByVal AppNumber As Long, idIcon As Long, idName As Long, idInfo As Long, UserData As Long)

This method is called when Windows wants to retrieve the resource identifiers of the icon, name and description of the applet. If there is more than one applet in your DLL, AppNumber specifies the applet number. You can set the UserData parameter to any value you choose – the value will be passed as a parameter to the CplDbClick method.

If the AppNumber value is 0, the hIcon, idName and idInfo parameters will be initialized to 1, 96 and 97, which are the values that are used by the Control Panel Applet Wizard. The only time you should change these values is if you wish for the system to not cache the icon and string information or if you are implementing more than one applet in an applet DLL. Caching is the preferred mechanism for control panel applets because it allows the system to display applet information without loading the applet. However, in cases where the name or icon of the applet needs to change each time it is displayed, caching must be disabled. In this case, set the parameters that you do not wish cached to the value of -1.

Important Note: The control panel applet framework and wizard only provide built in resources for one applet. The hIcon, idName and idInfo must be left at the default value (-1) for all applets other than the first one.

CplNewInquire

(ByVal AppNumber As Long, hIcon As Long, szName As String, szInfo As String, UserData As Long)

This method is called when Windows wants to retrieve the icon and name information for the applet. If there is more than one Applet in your DLL, AppNumber specifies the applet number (zero for the first applet). hIcon is the handle of the icon to use (you would typically store an icon in a picture box on a form and use the handle returned from the picture property if you wish to dynamically assign an icon in this manner). szName is the name of the applet (up to 31 characters) and szInfo the description of the applet (up to 63 characters). Text beyond the length specified will be ignored.

The hIcon, szName and szInfo parameters are initialized to the values set by the Control Panel Applet wizard for the first applet. If you are implementing more than one applet in an applet DLL, you must set these three parameters to the correct values for that applet.

You can set the UserData parameter to any value you choose – the value will be passed as a parameter to the CplDbClick method.

Important Note: The exact behavior of this method and the CplInquire method is inconsistent between operating systems. In other words, you cannot assume that these methods will be called, or what order they will be called in.

CplStartWParms

(ByVal AppNumber As Long, ByVal ExtraData As String) As Boolean

This method is similar to CplDbClick except that the ExtraData parameter is provided with additional parameters. This only applies to version 5.0 of shell32.dll.

CplStop

(ByVal AppNumber As Long, ByVal UserData As Long)

This method is called when an applet is about to be closed. You should perform any necessary cleanup operations here. If you have more than one applet in your applet DLL, the AppNumber parameter will indicate which applet is being stopped. After CplStop is called for all of the applets in your DLL, the CplExit method will be called.

Using Control Panel Applets with Services

The dwCPLServiceDemo example provided with the Toolkit illustrates how you can communicate with your service using the control panel applet. The OpenSCManager API function is used to access the service control manager. The OpenService API function then opens the service and returns a handle to the service. The ControlService API function can be used to pass values to a running service – calling this function ultimately leads to your services IdwEasyService_ OnUserControlCode method being called. You can, of course, also access your service via COM from the control panel applet.

A partial listing of the relevant code follows:

```
Private Sub Form_Load()  
    scHandle = OpenSCManager(vbNullString,  
vbNullString, _  
    SC_MANAGER_CONNECT)  
    If scHandle > 0 Then ' Success  
        hService = OpenService(scHandle, "dwBeeper", _  
SERVICE_USER_DEFINED_CONTROL Or  
SERVICE_INTERROGATE)  
        If hService > 0 Then  
            LblWarning.Visible = False  
        Else  
            cmd250.Visible = False  
            cmd1500.Visible = False  
        End If  
    End If  
End Sub
```

```

        End If
    Else
        MsgBox "Unable to open service control
manager"
    End If

End Sub

' cmd1500 and cmd250 are simple commands.
' In this case they set the beep duration to
' fixed values.
Private Sub cmd1500_Click()
    Debug.Print ControlService(hService, _
        USERIDFORSHORTERTIMEOUTVALUE, sv)
    Debug.Print Err.LastDllError
End Sub

Private Sub cmd250_Click()
    Debug.Print ControlService(hService, _
        USERIDFORSHORTESTTIMEOUTVALUE, sv)
    Debug.Print Err.LastDllError
End Sub

```

Installing and Testing Your Control Panel Applet

It is very important that your control panel applet DLL be available on the system when your control panel applet CPL file is installed. The best way to do this is to compile and register your applet DLL before you install the CPL file created with the Control Panel Applet Wizard.

You can also run your applet project in the VB environment before installing the CPL file, and any time afterwards when debugging the applet.

To install the control panel applet CPL file, you can:

1. Copy it into the system directory (the only way to install it under versions of Windows before Windows 2000/XP).
2. In Windows 2000/XP, add the CPL file to the registry as shown in the next section.
3. If the applet DLL file is not available, you will see a warning and the CPL file will fail to load.

Installing the CPL File on Windows 2000/XP

The following excerpt from the Microsoft Platform SDK explains how to install a control panel applet on Windows 2000/XP. Note that despite what follows, you can install a control panel applet on Windows 2000/XP by copying it into the system directory for testing purposes on your development system (you should not attempt to distribute control panel applets that are installed in the system directory).

Every Control Panel application is a dynamic-link library. However, the DLL file must have a .cpl file name extension. For Windows 2000 and later systems, new Control Panel applications should be installed in the associated application's folder under the Program Files folder. The DLL's path must be registered in one of two ways:

- If the Control Panel application is to be available to all users, register the path on a per-computer basis by adding a REG_EXPAND_SZ value to the HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\ControlPanel\Cpls key, set to the DLL path.
- If the Control Panel application is to be available on a per-user basis, use HKEY_CURRENT_USER as the root key instead of HKEY_LOCAL_MACHINE.

The following two examples register the MyCplApp control panel application. The DLL is named MyCpl.cpl and is located in the MyCorp\MyApp application directory. The first registry entry illustrates per-computer registration, and the second illustrates per-user registration.

```
HKEY_LOCAL_MACHINE
  Software
    Microsoft
      Windows
        CurrentVersion
          Control Panel
            Cpls

MyCpl1 = "%ProgramFiles%\MyCorp\MyApp\MyCpl.cpl"
```

-or-

```
HKEY_CURRENT_USER
  Software
```

```
Microsoft
Windows
CurrentVersion
Control Panel
Cpls
```

```
MyCpl="%ProgramFiles%\MyCorp\MyApp\MyCpl.cpl"
```

Distributing Your Control Panel Applet

You must distribute three files with your control panel applet.

Your CPL file	This is the file created by the Control Panel Applet Wizard. You should install it in the system directory (pre-Windows 2000), or in an application directory (Windows 2000/XP, and be sure to edit the registry entries as described earlier).
Your ActiveX DLL file	This is the file you create to implement the control panel applet.
Cpapplet.tlb	This is the type library file used internally by the framework to marshal control panel applet interfaces. Install this in the same directory as the CPL file.

Installing and Distributing Your Service

Installing and distributing your service is similar to installing and distributing any Visual Basic program (in other words, not a simple prospect).

In addition to all of the usual issues involved in distributing a Visual Basic component (making sure you ship all the necessary dependent components and runtimes, and register them properly on installation), services have a few additional issues that you must consider.

Compiling Your Component

This is one area where the service framework actually makes life easier for you than a comparable standalone component. If you have created Visual Basic component before, you know how critical backward compatibility is, and how managing the versions and compatibility modes of your Visual Basic project are something to watch closely.

Those issues remain important if you expose objects from your service using COM. However, the framework itself does not require that you maintain compatibility between versions. All invocations of your objects are based on the project name and not the class identifier (so be sure you don't use a project name that is already in use). Calls to your objects are early bound, but based on the interfaces in the `dwNTServ.tlb` type library that we provide.

Configuring Security

You may need to configure the security for your component depending on the type of service. These include:

- Set the Application (AppId) with appropriate launch and access security and impersonation levels.
- Set the login account (if not LocalSystem) to be able to log in as a service.
- Set security on individual COM objects to prevent unauthorized creation or access of those objects.
- Be sure the client system is set to an impersonation level of Impersonation, not Identity, if you wish to perform operations other than security tests in the context of the user.

There may be other securities to consider depending on your system and service.

Configuring Remote Systems to Access Objects From Your Service

Configuring client systems to access objects from your service through DCOM is the same as any Visual Basic ActiveX server accessed from DCOM. Refer to your VB or DCOM documentation for details. In brief, you must do the following:

- Distribute the VBR file created by the service configuration program.
- Distribute the dwNTServ.tlb type library and place it in the same directory as the VBR file.
- Use the CliReg32 application to register the objects for your service.
- Set the impersonation level of your client computer to “Impersonate” if your client object will be acting on behalf of the client.
- Be sure your client system has the necessary credentials to be authenticated on computer that the service is running on.
- Be sure your service is configured (using dcomcnfg) to allow launching and access to the client.

Refer to the section on security and impersonation for further details.

Service Executable Command Line Options

Your service executable supports the following command line options:

-RegServer [-Params <i>paramstring</i>] [-Silent]	Register your service to run in standalone mode with the simulator. <i>/Params paramstring</i> – Allows you to set a parameter string during installation that can be read at any later time. Enclose <i>paramstring</i> in double quotes if <i>paramstring</i> contains one or more space.
-UnRegserver [-Silent]	Unregister your service from standalone mode.
-I [-User <i>user</i> -Password <i>password</i>] [-Params	Install your service to run as a service. -User <i>username</i> – If specified, this overrides the user name provided by the service configuration file (if any).

<i>paramstring</i> [-Silent]	-Password <i>password</i> – If specified, this overrides the password provided by the service configuration file (if any). NOTE: The password is not tested for validity at install time. Use the –User and -Password options when you wish to set the service account during installation. These fields are not supported for services that are set to interact with the desktop. -Params <i>paramstring</i> – Allows you to set a parameter string during installation that can be read at any later time. Enclose paramstring in double quotes if paramstring contains one or more space. NOTE: If you had successfully installed a service and you make any service configuration changes, you must uninstall the service first before installing it again in order for those changes to take place.
-U [-Silent]	Uninstall your service from running as a service.
-V	Display the version of the service (in a message box).
-Sim	Run the executable as a standalone simulator.

The –Silent option prevents the display of message boxes during installation. Errors can be recorded to a log file, and if an error occurs it will be reflected in the exit code for the application.

NOTE: If you run a version 1.x service executable with any of the new command line options (user, password, params, and silent), no errors will be generated and those command line options will be ignored. The NT Service Toolkit’s Executable Launcher utility will detect this situation and produce an error message.

Redistributable Components

The following components can be distributed with your service.

- The service executable you create using the Service Configuration Program.

- The VBR file created for you by the Service Configuration Program.
- The dwNTServ.tlb type library (must be in the same directory as the service executable in order for it to run).
- The dwsvclnt.dll security component (must be registered).
- The dwbkthrd.dll background threading component (must be registered).
- The dwsock6.dll (must be registered), sockintf.dll and dwspyvb6.dll (must be registered) components for use with Winsock.
- The control panel applet CPL file created using the Control Panel Applet Wizard.
- The cpapplet.tlb type library (must be in the same directory as the CPL file in order for it to run).

Version Verification

We strongly recommend you use Desaware's VersionStamper to provide built-in component verification to your service. You can also implement remote monitoring and automatic updating of your service using VersionStamper.

Technical Support

Desaware prides itself on providing excellent technical support at no charge. At the same time, while we are glad to address any problems with our software, we know from experience that our software is often used in ways that we never imagined. As enabling technologies (i.e. technologies that allow VB programmers to do things that are beyond the typical VB application), we cannot characterize any of our components for every possible application.

In other words, while we will do our best to address any bugs in our products or issues that look like they have the potential of being bugs, we cannot write your code for you, or debug your program for you. Nor can we provide one on one consulting on particular applications.

When you contact us, we will assume that you are familiar with the material in this manual. We ask that you reduce any problems to the smallest set of code that duplicates the problem.

We cannot help you at all with system configuration problems, especially on the subjects of COM/DCOM and security.

Desaware cannot provide technical support on the issue of authentication of users or basic COM/DCOM functionality on your service's computer. Before you call us for technical support on subject related to client access, you must verify that you can create a standalone Visual Basic EXE on your service's computer, and access it via DCOM from the client system in question, where your client system is logged in under the same account that you wish to use to access the service. If you cannot do this, you have an authentication or connection problem that does not relate in any way to our toolkit. If you call us for support on what turns out to be an authentication problem, we will charge you our standard consulting rates for time spent on the problem. Authentication and account management issues must be resolved by your system administrator.

Framework Restrictions

The following are known restrictions of the Desaware NT Service Toolkit framework, as compared to creating a service from scratch using C++.

Configuration Issues

- You cannot use this service to create driver services (but you wouldn't use VB for drivers anyway, would you?).
- Services created with this framework always run as independent processes (most services should anyway). The framework only supports one service per executable (as do virtually all services).
- Services created with this framework do not support the `ERROR_SERVICE_SEVERE` and `ERROR_SERVICE_CRITICAL` options for dealing with startup errors. These options are suitable only for key driver and system components without which the system itself cannot operate properly.
- Detection of network binding service control manager events. These are only useful for network driver services.

Differences Between the Light Edition and Standard Edition of the Desaware NT Service Toolkit

The Desaware NT Service Toolkit is available in two editions. The “light” edition is included in SpyWorks Professional 6.2 and later. It is intended to replace the previous Visual Basic only service implementation provided in SpyWorks and provides roughly equivalent functionality to that approach but with considerably improved reliability and ease of use. The full Toolkit is sold as a standalone product, but is available at a substantial discount to SpyWorks Professional subscribers.

The following features from the toolkit are not available in the Light edition.

- Automatic background thread support for waiting on synchronization objects.
- Support for shared “application” objects that can be accessed by clients.
- Ability to expose client objects for access via COM or DCOM.
- OnParamChange method for detection of service parameter changes.
- OnHardwareProfileChange method for detection of hardware profile changes.
- OnDeviceEvent and RegisterDeviceEvent methods for detection of other device events.
- OnPowerRequest method for monitoring of automatic power management events in the service.
- No version 2.0 features are included in the light edition of the toolkit.

Other Sources of Information

Here are several other resources that we recommend for advanced Windows development.

www.desaware.com

Desaware's web site includes numerous technical articles on all aspects of Windows development. Be sure to also check the FAQ and support section for this product.

Dan Appleman's Visual Basic Programmer's Guide To The Win32 API

Written by Daniel Appleman (president of Desaware) and published by McMillan, (ISBN 0-672-31590-4) - this sequel to the original 16 bit API Guide applies the same philosophy to teaching the Win32 API to developers using Visual Basic and VBA based applications. With more examples, more functions, more tutorial style explanations and a full text searchable electronic edition on CD-ROM, this book should prove a worthy successor to the 16 bit API book. Covers Visual Basic version 4 through 6.

Available at most good bookstores, or directly from Desaware at a 20% discount - call (408) 404-4760 or email support@desaware.com.

An upgrade CD is available for owners of the "PC Magazine's Visual Basic Programmer's Guide to the Win32 API" ISBN: 1-56276-287-7 for \$24.99 + s&h directly from Desaware. Refer to our web site at www.desaware.com for additional information.

Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0: A Guide to the Perplexed

Written by Daniel Appleman (president of Desaware) and published by McMillan, (ISBN 1-56276-576-0) - this book is designed for those programmers interested in using Visual Basic's object oriented technology to develop ActiveX components including EXE and DLL servers, ActiveX controls and ActiveX documents. Unlike many books that simply rehash the Visual Basic documentation, this one serves as a commentary to clarify and extend the documentation. Of special interest to VersionStamper customers will be the chapters on OLE and COM technology that will help them further understand the process of registering components, and the chapters on versioning and licensing.

The VB6 version also includes two new chapters on IIS Application development.

Available at most good bookstores, or directly from Desaware at a 20% discount - call (408) 404-4760 or email support@desaware.com.

Moving to VB.Net: Strategies, Concepts and Code

Written by Daniel Appleman (president of Desaware) and published by Apress (ISBN# 1-893115-97-6).

VB.Net is not Visual Basic. COM+2.0 is not COM.

Porting is stupid. These are just a few of the things you'll learn as Dan takes you on a journey unlike any other into the world of VB.Net. He tackles strategic issues to help you determine when and whether to deploy VB.Net.

As always, Dan teaches the core concepts such as inheritance and multithreading, where VB6 programming habits can lead to costly design and development errors. And he covers the language changes to help you adapt to, and understand, this new environment.

Dan Appleman's Win32 API Puzzle Book and Tutorial for Visual Basic Programmers

Written by Daniel Appleman (president of Desaware) and published by Apress (ISBN# 1-893115-01-1). Appleman's Win32 API Guide covers 700 API functions. This book covers the other 7800. How? By teaching you everything you need to know to read and understand the Microsoft C documentation and create correct API declarations for use in Visual Basic. Presented in an entertaining puzzle/solution format that challenges you to solve real world API problems. In depth tutorials take you behind the scenes to understand what really happens when you call an API function from VB.

Windows API Online Help

The Professional Edition of Visual Basic includes Win31api.hlp and/or win32api.hlp - an online help reference for all API functions. These functions are declared in C and do not consider Visual Basic compatibility issues, however the information in chapter 3 of the Visual Basic Programmer's Guide to the Windows API (chapters 3 and 4 of the 32 bit book) will provide you with information on how to translate these functions to Visual Basic.

Microsoft's Developers Network CD Rom

This amazing CD-ROM and web site (<http://msdn.microsoft.com>) contain a wealth of information and sample code, plus the latest Visual Basic knowledge base.

Microsoft's Windows Software Development Kit and Win32 Software Development Kit

The sample code is all in C, but by the time you've read the Visual Basic Programmer's Guide to the Windows API or Win32 API, you'll know enough to be able to translate the C code to Visual Basic.

Index

- Account, 37
- AccountName, 116, 120
- Application Object, 72, 73
- Architecture, 21, 30
 - NT Service Toolkit, 69
- AutoStart, 33
- Background Tasks, 18
- Background Threads, 64, 94
- BackgroundExecute, 96
- BackgroundExecuteDelayed, 96
- BackgroundObject, 96
- BinaryPathName, 119
- Business Objects, 19
- ChangeServiceConfig, 117
- Checkpoint, 118, 119
- ClearWaitOperation, 51, 66
- Client Object, 74, 75, 77, 81
- Client Object Model, 77
- ClientExecuteBackground, 51, 80, 81
- CLSID, 90
- COM, 14, 18-24, 30, 51-53, 59, 69, 70, 71, 81, 83, 89, 92, 94, 100, 110, 143, 147, 153
- COM/DCOM, 85, 86, 151
- Command Line Options, 148
- Command Switches, 60
- Component
 - dwSCM, 111
- Components
 - Redistributable Components, 149
- Configuring, 32
- Control Object, 65
- Control Panel Applet, 136, 143
 - Installation, 145
 - Wizard, 136
- ControlsAccepted, 25, 26, 32-35, 45-47, 50, 118, 125
- ControlsAcceptedFlags, 118, 125
- ControlService, 116
- CplDbgClk, 140
- CplExit, 141
- CplGetCount, 141
- CplInit, 141
- CplInquire, 141
- CplNewInquire, 142
- CplStartWParms, 143
- CplStop, 143
- CreateService, 114
- CurrentState, 118, 119
- Dan Appleman's Developing COM/ActiveX Components with Visual Basic 6.0, 23, 82
- Dan Appleman's Visual Basic Programmer's Guide to the Win32 API, 64
- Dan Appleman's Win32 API Puzzle Book and Tutorial for VB Programmers, 53
- database, 100
- DCOM, 14, 18, 22, 25, 37, 51-53, 59, 69, 71, 83, 86-89, 100, 103, 109, 110, 148, 153
- DcomCnfg, 86, 108
- Debugging, 106, 108
- DeleteService, 117
- Dependencies, 38, 120, 132
- Description, 17
- DisplayName, 113-120
- dwAsyncSocket, 131
- dwBackThread, 94-97, 100
 - Rules, 97
- dwFTPclient, 127, 132
- dwHostEnt, 131, 134
- dwHTTP10, 127, 132, 134
- dwNTServ.tlb, 3, 25, 31, 90, 147, 148, 150
- dwProtEnt, 131
- dwSCM
 - Architecture, 111
- dwSecurity, 85, 92
- dwServEnt, 131

- dwServiceConfig, 111, 114-121
- dwServiceObject, 111-117, 123, 125
 - ControlService, 116
 - StartService, 115
- DWSOCK
 - Architecture, 130
 - Error Values, 132
 - Examples, 133
- dwSock.dll, 132
- dwSocketInfo, 133
- dwSockets, 127, 130-133
- dwSocketUtil, 131-134
- dwspv.dll, 132
- dwTCP, 131
- dwUDP, 131
- EnumDependentServices, 117
- enumServiceControls, 25, 34
- EnumServicesStatus, 112
- EnumServiceStates, 125
- ErrorControl, 117, 119
- Errors
 - Service, 102
- event log, 54
- Event Log
 - EventID, 54
 - ReportEvent2, 54
- Examples, 99
 - Beeper, 99
 - ControlPanelApplet, 99
 - FileWatch, 100
 - Launcher, 100
 - RemoteUser, 100
 - ReportEvent2, 99
 - TinyWeb, 100
 - TinyWeb2, 100
 - Tracing, 99
- ExecuteBackground, 80, 81, 94-97
- FTP, 127-133
- GetAppObject, 71, 82
- GetDescription, 35
- GetDisplayNameFromServiceName, 113
- GetInteractiveUser, 52
- GetServiceNameFromDisplayName, 113
- GetServiceObject, 73, 74
- GetUserInfo, 93
- GetVersion, 36
- Global Variables, 81
- HTTP, 100, 129-134
- IdwEasyServConfig, 25, 26, 32-41, 48-50, 65
- IdwEasyServConfig_DefaultTimes, 48
- IdwEasyService, 27, 28, 40-50, 53, 66, 67, 72, 75, 80, 143
- IdwEasyService2, 48
- IdwService, 22
- IdwServiceClient, 74-81, 103
- IdwServiceConfig, 22
- IdwServiceCtl, 27, 28, 41-50, 65, 67, 69
- IgnoreStartupErrors, 36
- Impersonate, 92
- Impersonation, 83, 85, 92, 108, 147
 - Types, 84
 - Anonymous, 84
 - Delegate, 85
 - Identity, 84
 - Impersonate, 84
- Information
 - Other Sources, 154
- initialization file, 106
- InitializeSCManager, 112
- InstallParameters, 49
- InteractWithDesktop, 36
- Interface
 - IdwServiceCtl, 37
- IP Address, 129
- LaunchObject, 96
- LoadOrderGroup, 120
- LockServiceDatabase, 114
- Manager
 - EnumServicesStatus, 112
- Method
 - CreateService, 114
 - dwServiceObject, 115

- GetDisplayNameFromServiceName, 113
- GetInteractiveUser, 37
- GetServiceNameFromDisplayName, 113
- InitializeSCManager, 112
- LockServiceDatabase, 114
- OpenService, 113
- QueryLockStatus, 115
- UnlockLockServiceDatabase, 114
- Microsoft Message Queue, 100
- Monitors
 - System, 17
- Multi-threading, 94
- NT Service
 - Architecture, 30
 - Configuring, 32
 - Features, 14
 - Framework, 30
- Objects
 - Application, 72
 - Business, 19
 - Client, 74
 - dwAsyncSocket, 131
 - dwFTPclient, 127, 132
 - dwHostEnt, 131
 - dwHTTP10, 127, 132, 134
 - dwProtEnt, 131
 - dwSecurity, 92
 - dwServEnt, 131
 - dwSock, 131
 - dwSocketInfo, 133
 - dwSockets, 127, 130-133
 - dwSocketUtil, 131, 133, 134
 - dwTCP, 131
 - dwUDP, 131
 - RunningService, 70
 - Synchronization, 64
- OnConnect, 76, 78, 80, 81
- OnContinue, 42
- OnDeviceEvent, 28, 46, 53, 153
- OnDisconnect, 78, 80
- OnHardwareProfileChange, 28, 46, 153
- OnLogout, 48
- OnParamChange, 28, 45, 153
- OnPause, 27, 42, 48
- OnPowerRequest, 28, 47, 153
- OnShutdown, 27, 43, 72, 73
- OnStart, 27, 42, 72, 75
- OnStop, 27, 43, 48, 72, 73, 79, 80, 103
- OnTimer, 27, 47-49, 110
- OnUserControlCode, 27, 45, 143
- OpenService, 113
- Password, 38, 120
- Program
 - Batch Mode, 60
 - Command Switches, 60
- Properties
 - RetrieveMode, 135
 - SocketData, 133
- Property
 - dwServiceObject, 115
- QueryLockStatus, 115
- QueryServiceConfig, 116
- QueryServiceStatus, 116
- Redistributable Components, 149
- Reference Counting, 72, 75
- RegisterApplicationObject, 52, 69, 72
- RegisterClientObjectName, 52, 70, 75
- RegisterDeviceNotification, 46, 53
- ReportEvent, 54
- ReportEvent2, 54, 99
- Resource Pool, 19
- RetrieveMode, 135
- RevertToSelf, 92
- RunningService, 52, 70, 71, 77-82, 103
- Security, 83
- Service Configuration Program, 56, 60
- Service Configuration Wizard, 60
- Service Control Manager, 111
- Service Executable Launcher, 63
- SERVICE_AUTO_START, 33
- SERVICE_CONTINUE_PENDING, 42
- SERVICE_DEMAND_START, 33

SERVICE_DISABLED, 33
 SERVICE_PAUSE_PENDING, 42
 SERVICE_PAUSED, 42
 SERVICE_RUNNING, 42
 SERVICE_START_PENDING, 42
 SERVICE_STOP_PENDING, 43
 SERVICE_STOPPED, 43
 ServiceAccessRights, 113, 114, 123
 ServiceConfiguration, 25, 26, 30, 32, 39, 41, 57, 65, 69, 99
 ServiceControlConstants, 116, 124
 ServiceControlRights, 112, 122
 ServiceErrorControlTypes, 122
 ServiceHandle, 118
 ServiceName, 113, 114, 118
 ServiceProcessId, 26, 39
 ServiceSpecificExitCode, 118
 ServiceStartTypes, 119, 121
 ServiceStateConstants, 118, 125
 ServiceType, 112, 117, 119, 121
 SetWaitOperation, 51, 65-67, 109
 Simulator Mode, 107
 SocketData, 133
 Software Agent, 18
 Software License, 3
 SpyWorks Concepts
 Winsock, 127
 StartService, 115
 StartType, 117, 119
 StartupParameters, 49
 States
 SERVICE_CONTINUE_PENDING, 42
 SERVICE_PAUSE_PENDING, 42
 SERVICE_PAUSED, 42
 SERVICE_RUNNING, 42
 SERVICE_START_PENDING, 42
 SERVICE_STOP_PENDING, 43
 SERVICE_STOPPED, 43
 Transitions, 41
 StopService, 51
 svcHardwareProfile, 34, 50
 svcParamChange, 34, 50
 svcPauseAndContinue, 34, 50
 svcPowerEvent, 34, 50
 svcShutdown, 25, 34, 50
 svcStop, 25, 34, 50
 Synchronization Objects, 64
 System Monitors, 17
 TagId, 120
 Tasks
 Background, 18
 Technical Support, 151
 Testing, 106, 108
 COM, 110
 DCOM, 110
 Thread Pool, 70
 Threads
 Background, 64, 94
 Timeout, 47, 49, 51, 65, 66
 TimeOuts, 35
 Trace, 14, 15, 55, 106
 TraceLevel, 55, 106
 Tracing. See Trace.
 Tracing and Logging, 106
 Tutorial, 24
 Types, 17
 UnlockLockServiceDatabase, 114
 UnregisterDeviceNotification, 53
 UpdateTransitionTime, 41, 43, 50
 UserControl, 45
 Util, 133
 Variables
 Global, 81
 VBR File, 25, 59, 86, 90, 104, 148, 150
 WaitComplete, 27, 40, 48, 66, 67
 WaitHint, 119
 Win32ExitCode, 118
 Winsock, 14, 94, 127-133, 150, 163
 Internet/Intranet, 127
 Learning, 128
 Package, 127
 Wizard
 Batch Mode, 60

Command Switches, 60
WM_DEVICECHANGE, 46

WM_POWERBROADCAST, 47

Desaware Product Descriptions

Thank you for your purchase of this Desaware product. We have additional quality software to enhance your programming efforts. Please visit our web site at www.desaware.com for detailed descriptions and product demos.

SPYWORKS Standard 6/Professional 7.0

SpyWorks in a nutshell? Impossible!

You're going to want to download the SpyWorks demo to even begin to understand its capabilities. This product has been evolving for several years, and it includes so many features it's hard to know where to begin. SpyWorks is a VB power tool. When you need to override VB's default behavior or to extend VB's functionality, you will want to use SpyWorks.

Do *That* in Visual Basic??

Want to put VB to the test? Want to learn advanced programming techniques? Want to keep the productivity of VB and have the functionality of C++? SpyWorks contains the low level tools that you need to take full advantage of Windows. Here are just a few of the features of this multi-faceted software package. For instance, have you ever wanted to detect keystrokes on a system-wide basis or detect when an event occurs in another application or thread using subclassing or hooks? SpyWorks can help you solve these problems by letting you tap into the full power of the Windows API without having to be an expert. SpyWorks lets you export functions from VB DLL's so that you can create function libraries, control panel applets, and NT Services. With its ActiveX extension technology, you can call and implement interfaces that VB5 or 6 do not support. SpyWorks includes the Desaware API Class Library, which assists programmers in taking advantage of the hundreds of functions that are built into the Windows API. SpyWorks is available in either the Professional (Pro) or Standard edition.

The Professional Edition includes .NET support for keyboard hooks, window hooks and subclassing (including cross-task subclassing) with examples in both Visual Basic.NET and C#. Additionally, a WinSock component with comprehensive VB source code that gives you complete control for Internet/intranet programming.

Other features are the NT Service Toolkit *Light Edition*. This application is a subset of the Desaware NT Service Toolkit product. It allows a developer to create true NT services using Visual Basic. A background thread component that allows you to easily create objects that run in a separate background thread.

It also contains extensive sample code and three product updates.

- The Professional Edition includes the Winsock Library, NT Service support and many other additional features & samples, plus three free updates. SpyWorks 2.1 (VBX Edition) is included in the Pro Edition.
- SpyWorks Standard is a subset of Professional. A feature comparison is available on our web site.
- Supports VB 4, 5 & 6, Windows 95, 98, 2000, NT and ME depending upon which version (or edition) of SpyWorks.

STATECODER 1.0

A .NET class framework that makes it easy to create and support powerful state machines using VB .NET or C#. Dramatically improves the reliability of applications, components and services that make use of the multithreading and asynchronous features of .NET.

VERSIONSTAMPER 6.5

Distributing Component-Based Applications? Beware DLL HELL!

You've distributed your application and it's working fine. But your end user is still in charge of their system. What happens when they install a program that overwrites a component that your software needs to run? Can you verify that your users have the correct files required by your application? Can you really afford to spend two hours on the phone trying to figure out exactly what went wrong? Now you can easily avoid component incompatibilities by adding VersionStamper to your toolkit. It lets you check the versions of your program's components on your end user's system, and correct the problem.

You are in control!

DLL Hell is a big problem, and with VersionStamper you can be in control of how this problem is detected and corrected. You determine dependency scanning (file size, date, version or other parameter), how and when the dependency scanning is done (upon start up, at midnight, at user's discretion), and how you want the problem resolved (automatically, an email message to your help desk, from a dependency list on your web site and more). This means you can handle versioning problems as simply as using a message box to call tech support, or even automatically updating the invalid components over the internet or corporate network. Imagine your application updating itself without user (or programmer) intervention! Imagine the hours and money saved in tech support calls! You can even use VersionStamper for incremental updates and bug fixes.

Is This For Real?

No, you don't have to pay a fortune in distribution fees - there are no run-time licensing fees. VersionStamper comes with a great deal of sample code. Don't distribute a component-based application without it!

- Checks the versions of your dependent files and notifies you or the user of potential problems.
- Internet extensions allow you to update versions across the Internet/intranets.
- Cool and USEFUL sample programs show you how it works.

Includes VB source code for the VersionStamper components that you can use in your applications.

NT SERVICE TOOLKIT 2.0

Create a fully featured service in minutes using Visual Basic – even debug your service using the Visual Basic environment! Supports all NT service options and controls. Adheres to all Visual Basic threading rules. Background thread support allows easy waiting on system and synchronization objects. Client requests supported on independent threads for excellent scalability, with client impersonation available allowing services to act on behalf of clients in their own security context. Client requests and service control possible via COM/COM+/DCOM.

Simulation mode for testing as an independent executable. Create control panel applets for service control and other purposes.

DESAWARE EVENT LOG TOOLKIT 1.0

Visual Basic allows you to log events to the NT/2000 event log, but does not allow you to create custom event sources - so every event belongs to the application VB runtime, descriptions are limited, and event categories unavailable. Even if you use the API to log events, creating custom event sources for your application is not supported by VB, and is difficult with C++.

Desaware's new Event Log Toolkit makes creation of event sources easy, and provides all the tools needed to create and log custom events. Now your applications and services can support event logs in a professional manner, as recommended by Microsoft

STORAGETOOLS ver 3.0

StorageTools is your key to the OLE 2.0 Structured Storage Technology. Structured Storage allows you to create files that organize complex data easily in a hierarchical system. It is like having an entire file system in each file. OLE 2.0 takes care of allocating and freeing space within a file, so just as you need not concern yourself with the physical placement of files on disk, you can also disregard the actual location of data in the file. Additionally, with its support for transactioning you can easily implement undo operations and incremental saves in your application. StorageTools allows you to take advantage of the same file storage system used by Microsoft's own applications. In fact, we include programs (with Visual Basic source code) that let you examine the structure of any OLE 2.0 based file so that you can see exactly how they do it!

StorageTools includes registration database controls for Windows NT, Windows 2000/XP, Windows 95 & 98. Plus, a simple resource compiler (with source) so that you can create your own .RES files for use with Visual Basic and more. 16 & 32 bit COM/ActiveX and .NET.

New for version 3.0! StorageTools 3.0 includes .NET support for accessing OLE Structure Storage from .NET assemblies.

DESAWARE ACTIVEX GALLIMAUFRY Ver. 2

What is it?

gal·li·mau·fry (gâl'e-mô'frê) noun
plural gal·li·mau·fries
A jumble; a hodgepodge.

[French galimafrée, from Old French galimafree, sauce, ragout : probably galer, to make merry. See GALLANT + mafrer, to gorge oneself (from Middle Dutch moffelen, to open one's mouth wide, of imitative origin).]

(From The American Heritage® Dictionary of the English Language, Third Edition copyright © 1992 by Houghton Mifflin Company)

What does a Twain control, spiral art program, set of linked list classes, a quick sort routine, a hex editor and a myriad of other custom controls have in common?

They are all part of Desaware's ActiveX Gallimaufry.

You'll find most of these controls useful, the rest entertaining – but we guarantee that you'll find them all educational, because they come with complete Visual Basic 6.0 source code.

Curious?

Want to learn some advanced API programming techniques? Visit our web site for a full description and demo.

THE CUSTOM CONTROL FACTORY V 4.0

The Custom Control Factory is a powerful tool for creating your own animated buttons, multiple state buttons, toolbars and enhanced button style controls in Visual Basic and other OLE control clients, without programming. With 256 & 24 bit color support, automatic 3D backgrounds, image compression, over 50 sample controls and more. Plus MList2 - an enhanced listbox control. 16 & 32 bit ActiveX controls and 16 bit VBXs included.