The Desaware
NT Service Toolkit
.NET Edition

Version 2.1

Windows 2000, XP and all .NET languages

by

Desaware, Inc.

Rev 2.1.0 (08/06)

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 © 2000-2006 Desaware, Inc. All rights reserved.".

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

Files: You may include with your program a copy of the files Desaware.ServiceToolkit.Interfaces.dll. NTServiceToolkitMM.msm, dwSCM.dll, or dwSCMNet.dll You may also distribute EXE, CPL, 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.

Limited Warranty

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

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective CD 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 Dr. Suite 48, San Jose, California 95136.

Table of Contents

Before You Begin

The Desaware NT Service Toolkit is available in three editions.

.NET Version	This edition of the toolkit is designed to make it easy to create powerful services in .NET, providing capabilities beyond those included in the .NET framework. It also allows for easy migration from the other editions of the toolkit. This edition is designed for use with Visual Basic .NET, C# or other .NET languages. It **does not** allow creation of services using Visual Basic 6.
COM (Full) Version	The COM based edition of the toolkit allows creation of full featured services using Visual Basic 6.
Demo Version (available for both COM and .NET editions)	Includes all features described in this documentation except for the service configuration wizard 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 EasySvnt.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. The demo version has an expiration date. Once it expires you can download a new version from Desaware's web site.

Introduction

The Desaware NT Service Toolkit is designed to make it easy to create reliable and supportable 2000/XP Windows services using Visual Basic .NET, C# and other .NET languages. It provides numerous features beyond those offered by the .NET framework, leading to significant cost savings.

The Desaware NT Service Toolkit for .NET offers the following unique features:

- Built-in service simulator makes it possible to test and debug services without actually installing them as a service. This not only speeds development, but is especially critical for testing startup and shutdown scenarios that are difficult to debug.
- Create full featured control panel applets using Visual Basic .NET, C#, and other .NET languages.
- Services created with this toolkit are entirely self-installing using the command line. There is no need for a separate "installer" tool.
- Command line installation allows specification of user, password and install time parameters – ideal for supporting customized automated installations during deployment.
- Services may automatically and simultaneously expose internal objects via both .NET remoting and COM/DCOM. Support for both remoting schemes is ideal for incremental migration – you can move your service to .NET without migrating all of your clients.
- Objects exposed via both COM and .NET remoting automatically receive service start and stop notifications directly from the framework.
- The service framework is fully instrumented for tracing and debugging, essential for rapid solution to problems both during development and after distribution.
- A high degree of compatibility with the COM version of the toolkit makes migration of services to .NET remarkably easy.

- Built in timer, support for background waits on .NET synchronization (WaitHandle) objects, and asynchronous operations on remotable objects created by clients. Built in synchronization makes it easy to avoid many of the synchronization problems associated with multithreaded applications.
- COM interop supported using a private thread-pool for maximum scalability regardless of whether client access is via COM or .NET remoting.
- Interactive service features including the ability to detect the logged on user, and when a user is logging off the system.
- Integration with Desaware's StateCoder™ makes it easy to create sophisticated and scalable state machine based services.

In addition to the above features (which are unique to the toolkit and represent capabilities beyond the simple framework included with .NET), this toolkit supports the following essential service tasks:

- True 2000/XP service allows detection and response to all service handler requests.
- Supports latest Windows 2000/XP features.
- Service controlled state transition timeouts (start, stop, pause, continue).
- Support for impersonation of clients (act on behalf of clients).
- Launch arbitrary background threads for asynchronous operations.

Reading the Documentation

This manual is intended to simultaneously support two audiences, readers using the toolkit for the first time, and those who are migrating from the COM version of the toolkit.

Information relating to migration from VB6 COM based services will appear in shaded blocks.

New Features for Version 2.0

If you are migrating your VB6 COM service directly from version 1.1 of the Desaware NT Service toolkit, here is a list of the features added to the toolkit for version 2.0. These features are supported both in the 2.0 COM edition toolkit, and in this .NET edition toolkit.

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 (see following section).

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

GetStateCoderMessageSource – Obtain a StateCoder message source for use when implementing state machines using Desaware's StateCoder in a service. This message source can notify your state machine when service based events occur.

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 if an interactive user is logged on and retrieve their account name and domain in most cases.

Improved Error Handling

Robust trapping and detection of runtime errors that occur in your Visual Basic component allow 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 Visual Basic 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.

What is an NT/2000/XP Windows Service?

A Windows 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?

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 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 services for many different reasons. Ultimately, a service has only three real advantages over a regular executable:

- A service can be configured to start 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 on 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

Windows 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, changes to the registry, the termination of running applications, and many more events.

Managing large numbers of synchronization objects and background threads can, however, be a very complex task – especially on server based applications where you have multiple clients. Not only do you have the normal array of synchronization issues involved in any multithreaded application, but services have unique problems owing to the fact that they may be paused or shutdown.

The Desaware NT Service Toolkit addresses this issue in two ways. First, it provides automatic monitoring of any synchronization objects you choose on a background thread that it both manages, and synchronizes to the primary service thread. Second, it provides integration with Desaware's StateCoder which provides an effective way to process messages based on synchronization objects, and service messages provided by the service framework.

Background Tasks

Consider a business object that runs on a server. If you implement it as a web service or remotable object, the server will create an object 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 when the system boots at a more convenient time. An alternate approach is to have the service launch a separate server process 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 server process.

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 .NET remoting, 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. You can use a technique called impersonation to act as 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 .NET's strength in creating components - in fact, all you need to do to create your service is to create a new assembly 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, VB and .NET

Traditionally, 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 the service framework provided with .NET simply maps those requirements to the .NET programmer, making thread synchronization a concern and potential source for subtle errors in all but the most simple services.

The earlier (COM based) version of this toolkit included a sophisticated thread management scheme to assure synchronization between theses threads. This was necessary because VB6 uses the STA (Single Threaded Apartment) model of threading, and simultaneous access to COM objects from different threads is simply not allowed.

This architecture has been preserved for the .NET edition. By providing automatic thread synchronization for most tasks, the framework eliminates many potential sources of bugs, and simplifies the design requirements, thus reducing development cost.

The Desaware NT Service Toolkit

This toolkit provides a framework that appears to the operating system as a 100% standard service, and simultaneously appears to a .NET component as a standard client application.

The approach follows from the following chain of reasoning:

- The .NET framework for creating services using Visual Basic .NET and C# is limited in features and is risky to use because of threading issues.
- Visual Basic .NET and C# are outstanding tools for creating components.
- A service executable can both use and expose .NET components.

- .NET 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 .NET 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 primary 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 component that you create using Visual Basic .NET, C# (or other .NET languages) 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, IdwEasyServConfig for configuration, and IdwEasyServ to control the service itself. Your component can be easily debugged using the Visual Studio .NET development environment either while running as a service or when running under the built in simulator. In addition, you can expose an application object that is shared by all service clients, and additional client objects.

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

Learning More

The documentation provided here is intended to address the needs of most programmers who wish to write services using the framework. However, writing services touches on virtually every aspect of Windows programming including but not limited to, NET remoting, to COM and 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 .NET remoting or DCOM, and mention some of the issues that relate to use of remoting and DCOM in services. But you'll have to look elsewhere for in-depth information on configuring remoting and DCOM and how to configure them for use in your enterprise.

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

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

If you are moving to .NET from VB6, we strongly encourage you to read at least the first two parts of Dan Appleman's book "Moving to VB.Net: Strategies, Concepts and Code". 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

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

Service Component Name

This is the name of the .NET assembly DLL that the service will load. This name should also represent the root namespace of the assembly you will create. If you decide later that you want to change the assembly name, you will have to run this configuration program again.

You should choose an assembly name that is unique – duplication of assembly 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

If your service will expose objects to COM or DCOM clients, 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. Set the Thread Pool size to 1 if you do not plan to expose objects to COM or DCOM clients.

Create Remoting Files

This option will cause your service to create default remoting configuration files for your service and for clients accessing the service through .NET remoting. You will need to modify these files to suit your own needs as described later in this documentation and in the .NET documentation.

This option will also create a VBR file which clients can use to access objects exposed by your service 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 Assembly DLL

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

Using the Add References dialog, add a reference to the "Desaware.ServiceToolkit.Interfaces" assembly. This assembly should appear in the .NET tab of the Add References dialog. This assembly is installed into the GAC (Global Assembly Cache) and is located in the NT Service Toolkit's "bin" directory on your system. This assembly must be distributed with your service. A merge module for this assembly is provided (also in the "bin" directory) and may be used for the distribution of this file.

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.vb or ServiceCfg.cs from the NT Service Toolkit's "Template" directory).

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.

GetDescription	This allows you to specify a name and description (Windows 2000 and XP) for your service. The service name and description appear in the service's control panel applet.
GetVersion	This allows the service executable to obtain the current version number from your component.

VB .NET

```
Imports Desaware.ServiceToolkit

Public Class ServiceConfiguration
    Implements IdwEasyServConfig

Private Function IdwEasyServConfig_AutoStart() _
 As Boolean Implements IdwEasyServConfig.AutoStart

End Function

Private Function _
 IdwEasyServConfig_ControlsAccepted() As _
 ServiceControls Implements _
 IdwEasyServConfig.ControlsAccepted
        Return ServiceControls.svcStop
End Function

Private Sub IdwEasyServConfig_DefaultTimes(ByRef _
 DefaultStartTime As Integer, ByRef _
 DefaultStopTime As Integer, _
 ByRef DefaultPauseTime As Integer, _
 ByRef DefaultContinueTime As Integer) Implements _
 IdwEasyServConfig.DefaultTimes

End Sub

Private Function IdwEasyServConfig_GetDescription _
 () As String Implements _
 IdwEasyServConfig.GetDescription
        Return "Your service display name here" +
vbNullChar + _
 "Your service description here"
End Function

Private Sub IdwEasyServConfig_GetVersion(ByRef _
 MajorVersion As Integer, ByRef MinorVersion As _
 Integer) Implements IdwEasyServConfig.GetVersion
 MajorVersion =  System.Diagnostics.
```

```
 FileVersionInfo.GetVersionInfo( _
 System.Reflection.Assembly.GetExecutingAssembly.
 Location).FileMajorPart
 MinorVersion = System.Diagnostics.FileVersionInfo.
 GetVersionInfo( _
 System.Reflection.Assembly.GetExecutingAssembly.
 Location).FileMinorPart
End Sub

Private Function _
 IdwEasyServConfig_IgnoreStartupErrors() As _
 Boolean Implements _
 IdwEasyServConfig.IgnoreStartupErrors

End Function

Private Function _
 IdwEasyServConfig_InteractWithDesktop() As _
 Boolean Implements _
 IdwEasyServConfig.InteractWithDesktop

End Function

Private Function _
 IdwEasyServConfig_ServiceAccount() As _
 String Implements IdwEasyServConfig.ServiceAccount

End Function

Private Function _
IdwEasyServConfig_ServiceAccountPassword() As _
 String Implements _
 IdwEasyServConfig.ServiceAccountPassword

End Function

Private Function _
 IdwEasyServConfig_ServiceDependencies() As _
 String Implements _
 IdwEasyServConfig.ServiceDependencies

End Function
End Class
```

C#

First, add the following line at the top of the configuration file:

```
using Desaware.ServiceToolkit;
```

Then add the following class to the file:

```
public class ServiceConfiguration: IdwEasyServConfig
{

public bool AutoStart()
 {
  return false;
 }

public ServiceControls ControlsAccepted()
 {
  return ServiceControls.svcStop;
 }

public void DefaultTimes(ref int DefaultStartTime,
 ref int DefaultStopTime, ref int DefaultPauseTime,
 ref int DefaultContinueTime)
 {
 }

public string GetDescription()
 {
  return "Your service display name\0
          Your Service Description";
 }

 public void GetVersion(ref int MajorVersion,
 ref int MinorVersion)
 {
  MajorVersion =  System.Diagnostics.
  FileVersionInfo.GetVersionInfo(
  System.Reflection.Assembly.
  GetExecutingAssembly().Location).
  FileMajorPart;
  MinorVersion =  System.Diagnostics.
  FileVersionInfo.GetVersionInfo(
  System.Reflection.Assembly.
  GetExecutingAssembly().Location).FileMinorPart;
 }

public bool IgnoreStartupErrors()
 {
  return false;
 }

public bool InteractWithDesktop()
 {
  return false;
 }
```

```
public string ServiceAccount()
 {
  return null;
 }

public string ServiceAccountPassword()
 {
  return null;
 }

public string ServiceDependencies()
 {
  return null;
 }
}
```

Note: Be sure to verify the return values for each interface method you implement. If you use the Visual Studio wizard to add interface implementation, you'll find the default return values it chooses are often incorrect.

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.vb or Service.cs from the NT Service Toolkit's "Template" directory):

VB .NET

```
Imports Desaware.ServiceToolkit

Public Class Service
 Implements IdwEasyService

Private Function IdwEasyService_OnContinue( _
 ByVal ControlObject As IdwServiceCtl) As Integer _
 Implements IdwEasyService.OnContinue

End Function

Private Function IdwEasyService_OnDeviceEvent( _
 ByVal ControlObject As IdwServiceCtl, ByVal _
 EventType As Integer, ByVal EventData As Integer) _
 As Boolean Implements IdwEasyService.OnDeviceEvent

End Function

Private Function _
 IdwEasyService_OnHardwareProfileChange( _
 ByVal ControlObject As IdwServiceCtl, ByVal _
 ChangeType As Integer, ByVal ChangeData As _
 Integer)  As Boolean Implements _
```

```
 IdwEasyService.OnHardwareProfileChange

End Function

Private Sub IdwEasyService_OnParamChange(ByVal _
 ControlObject As IdwServiceCtl) Implements _
 IdwEasyService.OnParamChange

End Sub

Private Function IdwEasyService_OnPause(ByVal _
 ControlObject As IdwServiceCtl) As Integer _
 Implements IdwEasyService.OnPause

End Function

Private Function IdwEasyService_OnPowerRequest( _
 ByVal ControlObject As IdwServiceCtl, ByVal _
 APMMessage As Integer, ByVal Flags As Integer) _
 As Boolean Implements IdwEasyService.OnPowerRequest

End Function

Private Function IdwEasyService_OnShutdown(ByVal _
 ControlObject As IdwServiceCtl, ByRef StopPending _
 As Boolean) As Integer Implements _
 IdwEasyService.OnShutdown

End Function

Private Function IdwEasyService_OnStart(ByVal _
 ControlObject As IdwServiceCtl) As Integer _
 Implements IdwEasyService.OnStart

End Function

Private Function IdwEasyService_OnStop(ByVal _
 ControlObject As IdwServiceCtl) As Integer _
 Implements IdwEasyService.OnStop

End Function

Private Sub IdwEasyService_OnTimer(ByVal _
 ControlObject As IdwServiceCtl) Implements _
 IdwEasyService.OnTimer

End Sub

Private Sub IdwEasyService_OnUserControlCode(ByVal _
 ControlObject As IdwServiceCtl, ByVal ControlCode _
 As Short) Implements _
 IdwEasyService.OnUserControlCode

End Sub
```

```
Private Sub IdwEasyService_WaitComplete(ByVal _
 ControlObject As IdwServiceCtl, ByVal ThreadID As _
 Integer, ByVal CompletionType As Integer, ByVal _
 ObjectIndex As Integer) Implements _
 IdwEasyService.WaitComplete

End Sub
End Class
```

C#

First, add the following line at the top of the configuration file:

```
using Desaware.ServiceToolkit;
```

Then add the following class to the file:

```
public class Service: IdwEasyService
{
public int OnContinue(IdwServiceCtl ControlObject)
 {
  return 0;
 }

public bool OnDeviceEvent(IdwServiceCtl
 ControlObject, int EventType, int
 EventData)
{
  return false;
 }

public bool OnHardwareProfileChange(IdwServiceCtl
 ControlObject, int ChangeType, int ChangeData)
{
  return false;
 }

public void OnParamChange(IdwServiceCtl
 ControlObject)
 {
 }

public int OnPause(IdwServiceCtl ControlObject)
 {
  return 0;
 }

public bool OnPowerRequest(IdwServiceCtl
 ControlObject, int APMMessage,
 int Flags)
 {
  return false;
```

```
 }

public int OnShutdown(IdwServiceCtl ControlObject,
 ref bool StopPending)
 {
  return 0;
 }

public int OnStart(IdwServiceCtl ControlObject)
 {
  return 0;
 }

public int OnStop(IdwServiceCtl ControlObject)
 {
  return 0;
 }

public void OnTimer(IdwServiceCtl ControlObject)
 {
 }

public void OnUserControlCode(IdwServiceCtl
 ControlObject, short ControlCode)
 {
 }

public void WaitComplete(IdwServiceCtl
 ControlObject, int ThreadID, int CompletionType,
 int ObjectIndex)
 {
 }
}
```

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:

- Be sure that your service executable is in the same directory as your component build directory (the destination directory for the built DLL).
- Register the service executable by running the program in a command line using the parameters "–RegServer".

- In your Visual Studio project, select the Debug configuration as the Active configuration. Bring up the Property Pages for your project and select the Configuration Properties – Debugging set. Under ‘Start Action’, set your service executable as the external program to debug. Then set the command line arguments to –Sim.
- Start debugging under Visual Studio.

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”.

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

Migrating a Service from VB6

Migrating an existing VB6 service to VB .NET is a relatively simple process. Simple, in that the two frameworks are remarkably similar. However, migration of any VB6 project to .NET is rarely trivial, so your own code may require substantial changes that have nothing to do with services at all.

Step 1 – Migrate Your VB6 Project

Open the VB6 ActiveX DLL project that contains your service component in Visual Studio .NET. This will bring up the VB .NET upgrade wizard. Have it save the resulting project in a directory of your choice.

Step 2 – Turn on Option Strict

Change the line at the top of all your code files from Option Strict Off to Option Strict On. You don't have to do this for the toolkit to work, but you want to for numerous reasons relating to code reliability.

Step 3 – Remove the Reference to EASYSERVLib

In the References tab of your solution explorer, or project references dialog, you'll see that your project has a reference to EASYSERVLib (Interop.EASYSERVLib.dll). You should remove this reference.

Step 4 – Add a Reference to Desaware.Service-Toolkit.Interfaces

This is the .NET assembly that defines the .NET interfaces that you will be using that replace the old COM interfaces.

Step 5 – Add an ' Imports' Statement to Your Files

Add the following line to your service configuration class, service class, and any application or client classes if present:

```
Imports Desaware.ServiceToolkit
```

Step 6 – Search and Replace

You can use the global search and replace to clean some of the COM interface artifacts.

Delete all occurrences of "**EASYServLib.**" by replacing it with an empty text.

Replace all occurrences of **enumServiceControls** with **ServiceControls**.

Fix the ServiceControl enumeration fields. If you see a field beginning with __Midl, delete everything except the final value and add ServiceControls.

For example: Replace **__MIDL___MIDL_itf_EasyServ_0000_0001.svcStop** with **ServiceControls.svcStop**

Step 7 – Remove the ServiceProcessId Method

Remove the ServiceProcessId function from your service configuration file. It is not used in the .NET edition of the toolkit.

Step 8 – Miscellaneous

There are a number of service control object methods that have different parameters with the .NET version of the toolkit. These will appear as compilation errors. Refer to the migration notes for each method.

Migrated classes will have a ProgID attribute. You can leave this in place or delete it as you wish.

You should, of course, set the assembly attributes to suit the needs of your service.

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 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.

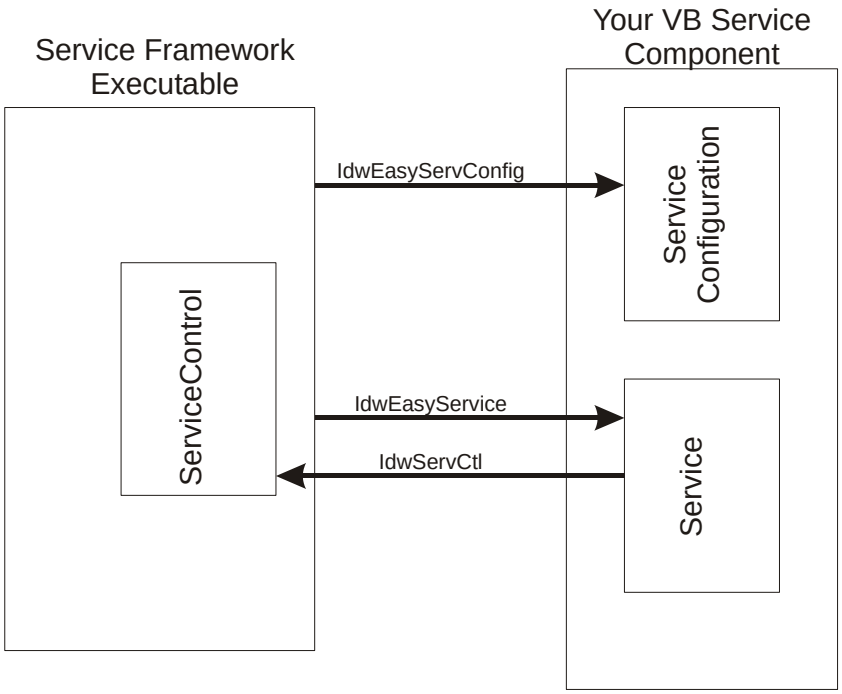

Figure 1
Core Section of the Service Framework

In each case, communication between the component and the service takes place on a private interface that is defined in the file Desaware.ServiceToolkit.Interfaces.dll which was installed when you installed the Toolkit (you will need to distribute this file with your service as well). 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 and then reinstall the service 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:

VB

```
Public Class ServiceConfiguration
Implements IdwEasyServConfig
```

C#

```
public class ServiceConfiguration: IdwEasyServConfig
```

Summary

Class name: ServiceConfiguration

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

Return Type: Boolean / bool

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:

VB: `Return(True)`
C#: `return(true);`

Important Notes: On a system using NT 4.0 Service Pack 5 or later, returning 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 dependencies as part of the return string for the IdwEasyServConfig_ ServiceDependencies function:

```
"Browser", "ProtectedStorage", "Replicator",
"RasMan" or "RasAuto"
```

ControlsAccepted

Return Type: ServiceControls

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 ServiceControls that defines the possible ways that the service control manager can interact with your service, and allows you to decide which controls to accept.

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).

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 ServiceControls enumeration.

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

VB:
```
Return(ServiceControls.svcStop Or
ServiceControls.svcPauseAndContinue)
```

C#:
```
Return(ServiceControls.svcStop |
ServiceControls.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

VB: DefaultTimes(ByRef DefaultStartTime As Integer, ByRef DefaultStopTime As Integer, ByRef DefaultPauseTime As Integer, ByRef DefaultContinueTime As Integer)

C#: void DefaultTimes(ref int DefaultStartTime, ref int DefaultStopTime, ref int DefaultPauseTime, ref int DefaultContinueTime)

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 Service Control object – this only sets the initial default value – the amount of time that 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

Return Type: String / 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:

VB:
```
Return("Place your display name here")
```
C#:
```
return("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.

VB:
```
Return("Place your display name here" + _
       vbNullChar + "Place your description here"
```
C#:
```
return("Place your display name here\0Place your
    description here");
```

GetVersion

VB: GetVersion(ByRef MajorVersion As Integer, ByRef MinorVersion As Integer)

C#: void GetVersion(ref int MajorVersion, ref int MinorVersion);

This method allows the service framework to obtain the version number of the service from the component. Add the following code to this method (VB & C# code is identical except for the trailing semicolon):

```
MajorVersion =
System.Diagnostics.FileVersionInfo.GetVersionInfo
(System.Reflection.Assembly.GetExecutingAssembly.
Location).FileMajorPart
MinorVersion =
System.Diagnostics.FileVersionInfo.GetVersionInfo
(System.Reflection.Assembly.GetExecutingAssembly.
Location).FileMinorPart
```

These values are displayed when you use the "–v" extension to display the version of an installed version.

IgnoreStartupErrors

Return Type: Boolean/bool

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:

VB:
```
Return(True)
```
C#:
```
return(true);
```

InteractWithDesktop

Return Type: Boolean/bool

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

VB:
```
Return(True)
```
C#:
```
return(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.
- 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.

Interactive services are known to pose a potential security risk – refer to the latest information on MSDN for details.

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

Return Type: String/string

This method allows you to set an account other than LocalSystem for your service. The majority of 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 remotely . 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 the following method:

VB: `Return("domain\userid")`
C#: `return("domain\\userid");`

Important Notes:

- Returning a result from this method will cause an error 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 when the service is being installed.

ServiceAccountPassword

Return Type: String/string

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

VB: `Return("password")`
C#: `return("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

Return Type: String/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:

VB: `Return("FirstService;SecondService")`
C#: `return("FirstService;SecondService");`

Note: Service dependencies are not used during system shutdown. In other words, you have no control over what order services are unloaded during shutdown, and services critical to your service may already be stopped by the time your service shutdown code is called.

ServiceProcessId

This method is used in the COM version of the toolkit and is not part of the IdwEasyServConfig interface in the .NET edition.

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. The framework releases its reference to your service object when the service stops. You should perform any necessary cleanup during the OnStop method rather than relying on finalizers.

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:

VB:
```
Public Class Service
    Implements IdwEasyService
```

C#:
```
public class Service: IdwEasyService
```

Summary:

Class Name: Service

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.
Privacy Issues:	Do not expose this object via Remoting.

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.

Method calls by the framework onto your service object through this interface are automatically synchronized. Thus you need not worry about multithreading issues when dealing with these methods.

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

VB: OnContinue(ByVal ControlObject As IdwServiceCtl) As Integer

C#: int OnContinue(IdwServiceCtl ControlObject)

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

VB: OnPause(ByVal ControlObject As IdwServiceCtl) As Integer

C#: int OnPause(IdwServiceCtl ControlObject)

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

VB: OnStart(ByVal ControlObject As IdwServiceCtl) As Integer

C#: int OnStart(IdwServiceCtl ControlObject)

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

VB: OnStop(ByVal ControlObject As IdwServiceCtl) As Integer

C#: int OnStop(IdwServiceCtl ControlObject)

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

VB: OnShutdown(ByVal ControlObject As IdwServiceCtl, ByRef StopPending As Boolean) As Integer

C#: int OnShutdown(IdwServiceCtl ControlObject, ref bool StopPending)

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 upon 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 upon 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 information regarding pending work to be done next time the service initializes, on disk or in the registry.

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 your operating system (especially NT and 2000), since there are a number of outstanding bugs on 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

VB: OnUserControlCode(ByVal ControlObject As IdwServiceCtl, ByVal ControlCode As Integer)

C#: void OnUserControlCode(IdwServiceCtl ControlObject, int ControlCode)

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

(Not available in Windows NT)

VB: OnParamChange(ByVal ControlObject As IdwServiceCtl)

C#: void OnParamChange(IdwServiceCtl ControlObject)

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

(Not available in Windows NT)

VB: OnHardwareProfileChange(ByVal ControlObject As IdwServiceCtl, ByVal ChangeType As Integer, ByVal ChangeData As Integer) As Boolean

C#: bool OnHardwareProfileChange (IdwServiceCtl ControlObject, int ChangeType, int ChangeData)

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_ Controls-Accepted method, or ControlObject's IdwServiceCtl Controls-Accepted 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

(Not available in Windows NT)

VB: OnDeviceEvent(ByVal ControlObject As IdwServiceCtl, ByVal EventType As Integer, ByVal EventData As Integer) As Boolean

C#: bool OnDeviceEvent (IdwServiceCtl ControlObject, int EventType, int EventData)

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

(Not available in Windows NT)

VB: OnPowerRequest(ByVal ControlObject As IdwServiceCtl, ByVal APMMessage As Integer, ByVal Flags As Integer) As Boolean

C#: bool OnPowerRequest (IdwServiceCtl ControlObject, int APMMessage, int Flags)

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_POWERBROAD-CAST 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 feature extensions provided by the framework to authors of services using this Toolkit.

OnTimer

VB: OnTimer(ByVal ControlObject As IdwServiceCtl)

C#: void OnTimer(IdwServiceCtl ControlObject)

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.

Remember: calls to this method by the framework are synchronized by the framework with the other methods on the IdwEasyService interface.

Note: Do not allow the code in this event to take longer than the default Pause or Stop times set in the IdwEasyServConfig_ 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. This is a consequence of the automatic synchronization, that prevents the OnStop or OnPause method call from occurring until you exit this method call.

WaitComplete

VB: WaitComplete(ByVal ControlObject As IdwServiceCtl, ByVal ThreadID As Integer, ByVal CompletionType As Integer, ByVal ObjectIndex As Integer)

C#: void WaitComplete(IdwServiceCtl ControlObject, int ThreadID, int CompletionType, int ObjectIndex)

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 may be optionally implemented by a service object that is configured to interact with the desktop.

OnLogout

VB: OnLogout(ByVal ControlObject As IdwServiceCtl)

C#: void OnLogout(IdwServiceCtl ControlObject)

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.

Summary:

Class Interface:	IdwServiceCtl
Rules:	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/string)

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

StartupParameters (String/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 (Integer/int)

This value indicates the interval of the built in timer in milliseconds. Your service object's IdwEasyService_OnTimer method 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 (ServiceControls)

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 (Not available in NT 4).
SvcHardwareProfile = 0x20	Service accepts hardware profile changes (Not available in NT 4).
SvcPowerEvent = 0x40	Service accepts power events (Not available in NT 4).

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

VB: UpdateTransitionTime(ByVal Timeout As Integer)

C#: void UpdateTransitionTime (int Timeout)

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

VB: SetWaitOperation(ByVal WaitHandles As WaitHandle(), ByVal bWaitAll As Boolean, ByVal Timeout As Integer) As Integer

C#: int SetWaitOperation(WaitHandle[] WaitHandles, bool bWaitAll, int Timeout)

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.

Those migrating services from the COM edition of the toolkit will notice that the first parameter of this method has changed from an array of 32 bit handles (referring to Windows synchronization objects) to a .NET array of System.Threading.WaitHandle objects. Refer to the Launcher sample application for an example of how to create a class (GenericWaitHandle) that wraps a system handle.

ClientExecuteBackground

VB: ClientExecuteBackground(ByVal ServiceClient As IdwServiceClient)

C#: void ClientExecuteBackground (IdwServiceClient ServiceClient)

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 service objects for additional information on this method.

The COM edition used 32 bit values to identify clients because of the requirement under COM to be able to identify a client object without actually holding a reference to it (and marshaling it between threads). Under .NET, this is not necessary, so an actual reference to the client object, which implements the IdwServiceClient interface, is used.

ClearWaitOperation

VB: ClearWaitOperation(ByVal WaitThreadId As Integer)

C#: void ClearWaitOperation(int WaitThreadId)

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

Not available in NT 4

VB: GetInteractiveUser(ByRef domain As String) As String

C#: string GetInteractiveUser(ref string domain)

This function returns the user name of the user currently logged on. The Domain parameter returned will be set to the name of the domain of the user currently logged on. 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

VB: RegisterApplicationObject(ByVal AppObject As MarshalByRefObject)

C#: void RegisterApplicationObject (MarshalByRefObject AppObject)

This method allows you to set an object to be the shared application object for your service when accessed remotely. 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 for a detailed explanation of this method.

This method exposes the object for access both by COM/DCOM and .NET remoting clients.

Under .NET remoting, the object is registered as a Singleton object with the name Application.Soap.

Under COM/DCOM the object can be referenced using the GetApp-Object method of the application's RunningService object.

You must use a configuration file, or your own code, to register an appropriate channel in order for the object to be available through .NET remoting.

RegisterClientObjectName

VB: RegisterClientObjectName(ByVal ClientObjectName As String)

C#: void RegisterClientObjectName (string ClientObjectName)

This method allows you to specify the name of the object that the framework should create when users request client objects via COM or DCOM. The ClientObjectName parameter should be the class name of a MarshalByRef based class which you wish to expose to COM/DCOM clients.

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.

You need only use this method if you wish to expose objects to COM or DCOM based clients. Only one client object can be exposed through COM or DCOM (though they can create and expose others).

To expose objects for access via .NET remoting, configure the remoting configuration file for your service to specify those objects you wish to expose.

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

RegisterDeviceNotification

VB: RegisterDeviceNotification(ByVal NotificationFilter As Integer) As Integer

C#: int RegisterDeviceNotification (int NotificationFilter)

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).

The method automatically arranges 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 UnregisterDevice-Notification 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.

UnregisterDeviceNotifcation

VB: UnregisterDeviceNotification(ByVal DeviceNotificationHandle As Integer)

C#: void UnregisterDeviceNotification (int DeviceNotificationHandle)

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

ReportEvent

VB: ReportEvent(ByVal EventType As EventReportTypes, ByVal EventString As String, ByVal BinaryData As Byte())

C#: void ReportEvent (EventReportTypes EventType, string EventString, Byte[ ] BinaryData)

This method adds an event to the application event log.

The EventReportTypes 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

VB: ReportEvent2(ByVal Source As String, ByVal EventType As EventReportTypes, ByVal Category As Short, ByVal EventId As Integer, ByVal EventString As String, ByVal BinaryData As Byte())

C#: void ReportEvent2 (string Source, EventReportTypes EventType, short Category, int EventId, string EventString, Byte[] 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 EventReportTypes 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.

You can create custom, multilingual or self-installing event sources using Desaware's Event Log Toolkit.

Trace

VB: Trace(ByVal message As String, ByVal level As Integer)

C#: void Trace (string message, int level)

This method sends the specified string to all active trace listeners. 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 configuration file. Refer to the section on Testing and Debugging for more information on the tracing and logging capabilities of the framework.

Messages reported using this method will be written under the tracing category "EasySvNT".

In the COM version of the toolkit, all tracing goes to a log file. The .NET version uses the .NET framework capabilities. You can specify output to a log file, event log, or any other trace listener. Also, in the COM version, the tracing goes to a log file located in the same folder as your Service Executable. In the .NET version, if you output to a log file, you would need to include the path of the log file.

GetStateCoderMessageSource

VB: GetStateCoderMessageSource () As Object

C#: object GetStateCoderMessageSource ()

One of the new features of the .NET edition of the NT Service Toolkit is integration with Desaware's StateCoder for .NET. StateCoder allows you to implement sophisticated state machines in .NET, and is largely self-synchronizing, thereby helping to eliminate potentially subtle threading problem when using multithreading and asynchronous operations. StateCoder state machines use message sources to provide synchronized input to state machines. Message sources can reflect virtually any kind of event or information, from simple timeouts or end of asynchronous operations, to MSMQ message and system events.

The IdwServiceCtl.GetStatecoderMessageSource function allows you to obtain a StateCoder message source for use when implementing state machines in a service. This message source can notify your state machine when service based events occur. The SCMessageType enumeration defines the type of service based event that has occurred. These events correspond to the implemented methods in the IdwEasyService interface. This allows you, for example, to have your state machine receive a message when the service is paused, or stopped.

Refer to the StateCoder documentation for further information on StateCoder message sources.

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.

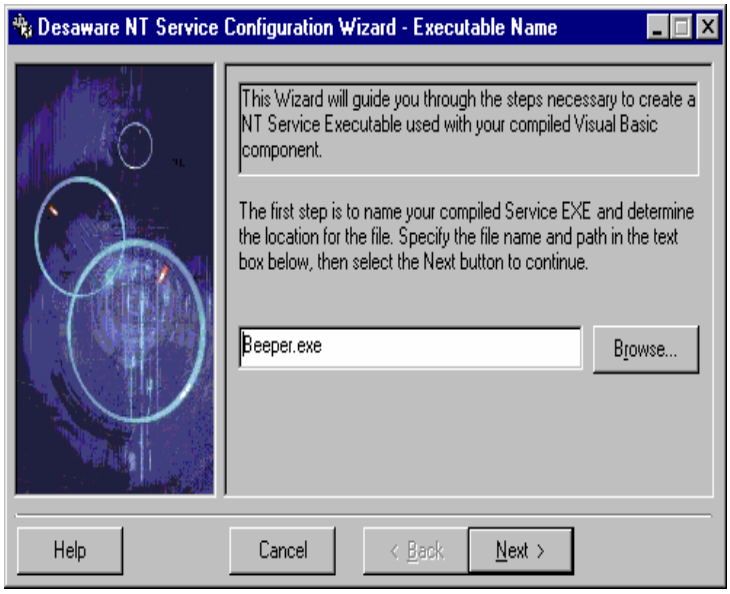

Figure 2
NT Service Configuration Wizard
Assigning Executable Name & Location

Assembly Name

Enter the full Assembly name of the .NET 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 assembly name, you will have to run this configuration program again.

It is important that you should choose an assembly name that is unique – the duplication of assembly 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 assembly name length can be up to 18 characters.

You must also choose the target .NET framework for your service.

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 other Windows applications. 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.

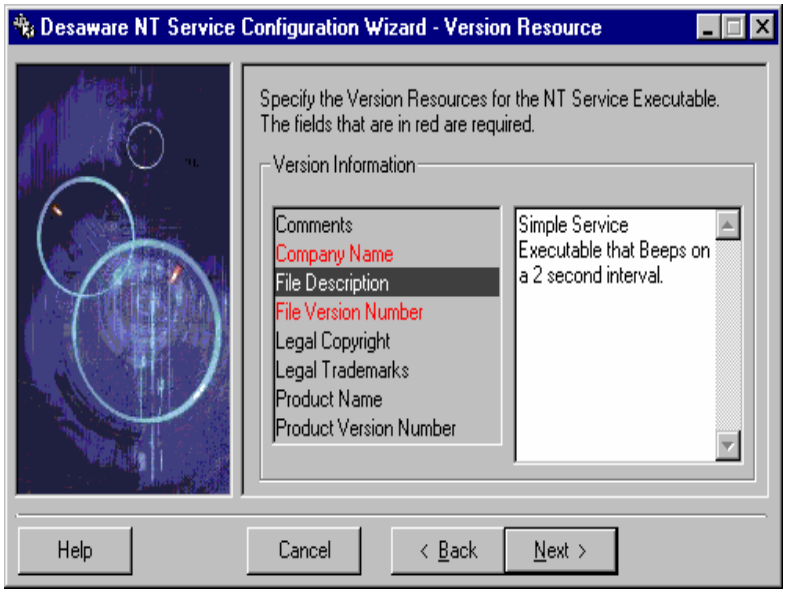

Figure 3
NT Service Configuration Wizard
Specifying Version Resources

Thread Count

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

If you are only exposing objects via .NET remoting, or not exposing objects you should set a thread count of 1. Otherwise, if your service will expose client objects for use by COM, we suggest setting a thread count of 4.

Create Remoting Files

This step allows you to create remoting or remote automation files for accessing objects exposed by your service.

The *Generate VBR file for DCOM* checkbox allows you to create a VBR file which is used by the clireg32 application to create the necessary registry entries for accessing your service objects remotely via COM or DCOM. The VBR file will be created in the same directory and have the same base name as your service executable.

To create a VBR file, check the *Generate VBR file for DCOM* checkbox and enter the description of the service object for your service in the *Service Object Description* text box.

If you are exposing service objects for DCOM, you must use the Service Configuration Wizard to create a corresponding VBR file for your Service EXE each time you recompile your Service EXE. The previous VBR file will only work with the previous Service EXE.

This is required because the VBR files contain GUIDs for the exposed RunningService object of your Service EXE. But, each time you recompile your Service EXE file, a new set of GUIDs are generated for the exposed RunningService object, therefore a new VBR file should also be created.

The Service Configuration Wizard will check the output folder of your Service EXE for a VBR file having the same name as your Service EXE (except with a .VBR extension). If the .VBR file exists, it will initialize the *Generate VBR file for DCOM* checkbox to the checked state and read the .VBR file's APPDESCRIPTION field into the *Service Object Description* text box. If you are exposing service objects for DCOM and you are creating your Service EXE in batch mode. Be sure a copy of the corresponding VBR file is present in the same folder as your output Service EXE file. Otherwise, a new VBR file will not be generated for the new Service EXE.

The *Generate Service Config file for Remoting* checkbox allows you to create default remoting configuration files which you can use as a template to specify which objects in your service you wish to remote as client activated objects. A remoting file will be created for the server (running the service). In most cases, you can also use the generated remoting file as a template file for the clients accessing the service.

To create remoting configuration files, check the *Generate Service Config file for Remoting* checkbox and enter the listening port number in the *Listening port number* text box. If your service will expose a client object, enter the name of the client object in the *Exposed client name* text box. **NOTE:** You may need to make additional modifications to the generated remoting configuration file before it can be used for your particular configuration. Refer to the MSDN documentation on .NET remoting for more details.

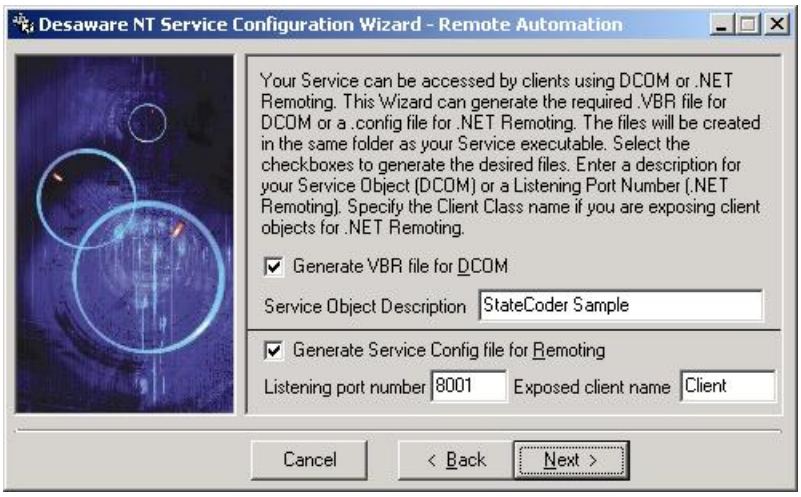

Figure 4
Remote Automation Dialog Box

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 .NET assembly containing the service objects is located in the same folder as your service executable prior to installing or registering your service executable. The file Desaware.Service-Toolkit.Interfaces.dll must be installed in either the same directory or the Global Assembly Cache. 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

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 upon an existing one). The following describes the batch command switches.

Command Switches

/batchbuild	This switch is necessary in order to run the NT Service Configuration Wizard in batch mode.
/infile=*filename*	This switch is necessary. The *filename* variable represents the existing Service Executable that the newly compiled Service Executable will be based on. The newly compiled Service Executable will use the same Service Assembly, retain the same Version Resource, and Thread Pool settings as the existing file. The filename variable should also include the path.
/outfile=*filename*	This switch is optional. The *filename* 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 /infile switch.
/fileversion=x.x.x.x	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 /fileversion 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 /infile 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 running the 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 exists. 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 beeperror.log file.

```
NTServiceWizard.exe /batchbuild /fileversion=2.0.1.0
/infile=c:\ntservtk\samples\beeper\beeper2.exe
/logfile=c:\ntservtk\samples\beeper\beeperror.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.. To use the Service Executable Launcher, just drag and drop your service executable from Windows 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 and then reinstall the service in order for those changes to take effect.

The Register ActiveX Component and Unregister ActiveX DLL buttons provide a convenient way to register or unregister COM components and are not used with your .NET service objects.

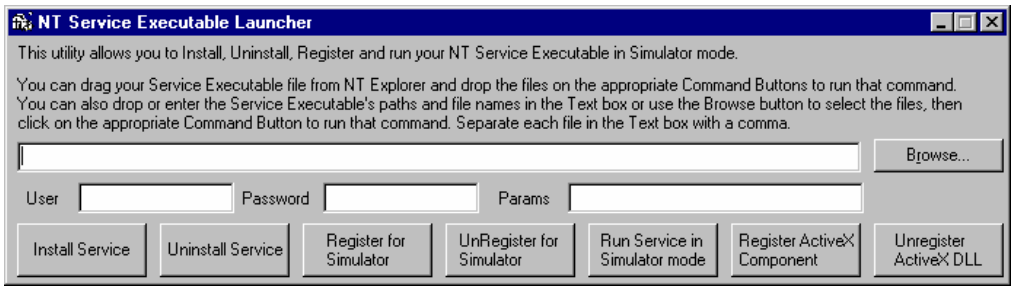

Figure 5
NT Service Executable Launcher Utility

Background Threads and Synchronization Objects

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, 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.

The .NET framework contains managed wrappers for many of these synchronization objects in the System.Threading namespace. Synchronization objects in .NET inherit from the System.Threading.-WaitHandle object.

An in-depth discussion of synchronization objects is beyond the scope of this manual. You can find an extensive discussion of synchronization objects in MSDN both under the Platform SDK (for API functions) and the documentation for the System.Threading namespace.

There are two ways to use synchronization objects. 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 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 automatically create background threads for this purpose.

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:

VB:
```
SetWaitOperation(ByVal WaitHandles As _
    WaitHandle(), ByVal bWaitAll As Boolean, ByVal _
    Timeout As Integer) As Integer
```

C#:
```
int SetWaitOperation(WaitHandle[] WaitHandles,
    bool bWaitAll, int Timeout)
```

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

The WaitHandles() array is an array of WaitHandle variables. The array should be declared for the exact number of object handles you will use. 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:

VB

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

C#

```
WaitHandle[] ObjectList = new WaitHandle[0];
WaitHandle[0] = yourhandle;
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.

VB: `ClearWaitOperation(WaitThreadId As Integer)`
C#: `ClearWaitOperation(int WaitThreadId)`

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 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:

VB:
```
WaitComplete(ByVal ControlObject As
IdwServiceCtl, ByVal ThreadID As Integer, ByVal
CompletionType As Integer, ByVal ObjectIndex As
Integer)
```
C#:
```
void WaitComplete(IdwServiceCtl ControlObject,
int ThreadID, int CompletionType, int
ObjectIndex)
```

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

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 either .NET remoting, COM or DCOM to obtain a reference to an object exposed by the service.

.NET Remoting vs. COM/DCOM

One of the key design goals of the .NET version of this toolkit is to make it as easy as possible to provide client access to objects in the service via both .NET remoting and COM/DCOM. This is in recognition of the fact that the transition to .NET will be a long process, and it will be very common to have a mix of COM and .NET based applications in many enterprises.

The COM remoting subsystem in this toolkit is identical to that in the COM edition of the NT Service Toolkit. It is designed to be called from VB6 and other COM based clients, and adheres to the COM STA (Single Threaded Apartment) threading model.

The .NET remoting subsystem uses .NET remoting. As such, it does not impose any threading restrictions on objects. This freedom does, however, impose on the programmer the responsibility to handle synchronization in any case where a client might access shared data.

.NET Remoting

A complete discussion of .NET remoting is beyond the scope of this document. Refer to the .NET documentation, or the excellent book Advanced .NET Remoting by Ingo Rammer (published by Apress: www.apress.com, available in both VB .NET and C# editions).

The following description assumes you have at least a basic familiarity with .NET Remoting.

The first, and most important thing to remember about .NET remoting with the NT Service Toolkit is this:

Services created with this toolkit are true Windows services, and as such, are fully capable of acting as hosts for .NET remotable objects.

In other words, services you create are able to correctly remote all types of .NET remotable objects (Singleton, Singlecall, and Client Activated).

If you do not wish to use the COM/DCOM subsystem, you should not use the IdwServiceCtl RegisterApplicationObject or RegisterClient-ObjectName methods. Simply define the objects you wish to remote, and register them for remoting using your services configuration file or the appropriate .NET framework functions.

Keep in mind that .NET remoting does not automatically synchronize access to objects. It is up to you to add thread synchronization as necessary. This is especially important for Singleton objects, which can be accessed simultaneously by multiple clients. It is also important for SingleCall or Client Activated Objects that reference shared data.

Note: Your service's Service Control object (IdwServiceCtl interface) is automatically synchronized.

.NET Remoting Configuration File

The NT Service Toolkit framework will automatically load a remoting configuration file if one exists with the following name:

yourservicedll.config

Where *yourservicedll* is the name of your service DLL file without the DLL extension.

For example: if your service DLL is myservice.dll, the configuration file name will be myservice.config.

You must include a remoting configuration file (or perform configuration in your service startup code) to use .NET remoting.

The Service Framework Object Architecture

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

1. Objects exposed only through .NET remoting.
2. Application objects exposed through both .NET remoting and COM.
3. Client objects exposed through both .NET remoting and COM.

Let's consider these types of objects.

Objects Exposed Only Through .NET Remoting

These objects were discussed earlier. Use standard .NET Remoting techniques to expose any MarshalByRef object.

Application Objects Exposed Through Both .NET Remoting and COM

You may register any one MarshalByRef object to serve as the Application object which will be accessible through both .NET remoting and COM. You create an instance of the object in your primary service object, and register it by calling the RegisterApplicationObject method on the ServiceControl object (IdwServiceCtl interface). The object may have any class name you choose – we use Application in our examples by convention only.

Calling the RegisterApplicationObject method causes two things to happen:

1. The object becomes available to COM clients via the GetAppObject method of the RunningService object (which will be discussed shortly).
2. The object is registered as a Singleton object via .NET remoting under the name Application.Soap.

When accessed via COM, all references to the application object will be synchronized and marshaled to the primary service thread. This is important to remember, because it means that long operations on this object can impair the performance of the entire service.

When accessed via .NET remoting, references to the application object will not be synchronized and will take place on the remoting thread.

This means it is important for you to perform any necessary synchronization on methods of your Application object.

Note to those migrating from the COM edition of the toolkit: Under COM, access to the Application object was automatically synchronized owing to the use of the STA COM threading model. Under .NET, the Application object is free threaded. You should review your design and add SyncLock (C# lock) statements to synchronize any methods or properties of this object where simultaneous access by different clients might lead to errors.

Refer to Dan Appleman's book "Moving to VB.Net: Strategies, Concepts and Code" for a discussion of .NET multithreading written for those migrating from VB6.

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.

Client Objects Exposed Through both .NET Remoting and COM

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

1. Long client operations should not impact the overall performance of the service.
2. Long client operations should have minimal impact on other clients.
3. 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.

Client objects created via .NET remoting are accessed from a thread pool provided by the .NET runtime. Client objects created via COM or DCOM run on a thread pool managed by the NT Service Toolkit framework. The number of threads in this threadpool is defined by the service configuration program when you create the service.

By using threadpools in this manner, 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 that inherits from MarshalByRefObject and registering the name of the class with the ServiceControl object using the RegisterClientObjectName method.

You may only register one object type for access by both COM and .NET. You may register additional object types for access only via .NET remoting using standard .NET remoting techniques.

The RunningService Object (COM Clients Only)

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 primary service object.

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. The class should inherit from MarshalByRefObject.
3. In your service object, create an instance of the application object, typically using code such as:

   VB: `Private appobject As New Application`
   C#: `Application appobject = new Application();`

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

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

If you wish your application object to be accessible via .NET remoting, add a remoting configuration file of the name *yourservice*.config, where *yourservice* is the name of your service DLL (without the DLL extension). See the next section for a sample configuration file. While you do need to specify a listening channel, you do not need to define the Application object – the RegisterApplicationObject method automatically registers your object under the name Application.Soap.

That's all there is to it.

In the COM edition of this toolkit, developers would have to use a variety of techniques to avoid circular reference problems both with Application objects, and with objects they reference (such as the service control object). With the .NET edition of the toolkit, this is not a concern, as the runtime will automatically clean up objects when they are no longer referenced.

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. The object must inherit from MarshalByRefObject.
3. Implement the IdwServiceClient interface using the following code:

   VB:
   ```
   Implements IdwServiceClient
   ```
   C#:
   ```
   yourclass: IdwServiceClient
   ```

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

   VB:
   ```
   ControlObject.RegisterClientObjectName _
     ("clientobjectname")
   ```
   C#:
   ```
   ControlObject.RegisterClientObjectName
     ("clientobjectname");
   ```

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

   If you wish the client object to be accessible via .NET remoting, add a remoting configuration file of the name *yourservice*.config, where *yourservice* is the name of your service DLL (without the DLL extension).

   Here is an example of a typical configuration file for a service:

```
<configuration>
  <system.runtime.remoting>
```

```
    <application name="dwEasyServ">
      <lifetime leaseTime="2M"
          renewOnCallTime="1M"/>
      <service>
        <activated type="dwEasyServ.Client,
          dwEasyServ"/>
      </service>

      <channels>
        <channel port="8001" ref="http" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

You should define each client activated object and define a default listening port for your service as well.

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.

When migrating a client object from the COM edition of the toolkit, be sure to remember that your client object must inherit from MarshalByRefObject if you wish it to be accessible via .NET remoting.

The IdwServiceClient Interface and Client Objects

The NT Service Toolkit provides a mechanism by which client objects can obtain important information during the time they exist. The IdwServiceClient interface allows client objects to obtain references to the underlying service object when the client is created. The interface also provides notification when a client has disconnected, and when the service is stopping. It also makes it easy to start asynchronous background operations on the client.

Client objects that are exposed using the RegisterClientObjectName method (under both COM and .NET remoting) are required to implement the IdwServiceClient interface.

However, other .NET remoting clients (in cases where you expose additional client activated objects) may implement this interface if they wish, in which case they too will receive the appropriate notifications.

Service Specific Issues Relating to Client Objects

In a normal component that exposes objects, the lifetime of the component is dictated by the life of the objects. Under .NET, object lifetime is specified by an activation lease. Under COM, lifetime is determined by reference counting. 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" or other 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. 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.

Those of you migrating from the COM version of the toolkit may recall a recommended design pattern to use to maintain collections of client objects in order to avoid circular reference problems. Under .NET, you need only add a client to a collection when the IdwServiceClient_OnConnect method is called, and remove it when the IdwServiceClient_OnDisconnect method is called.

Figure 6 illustrates the actual object model used when you expose a client object.

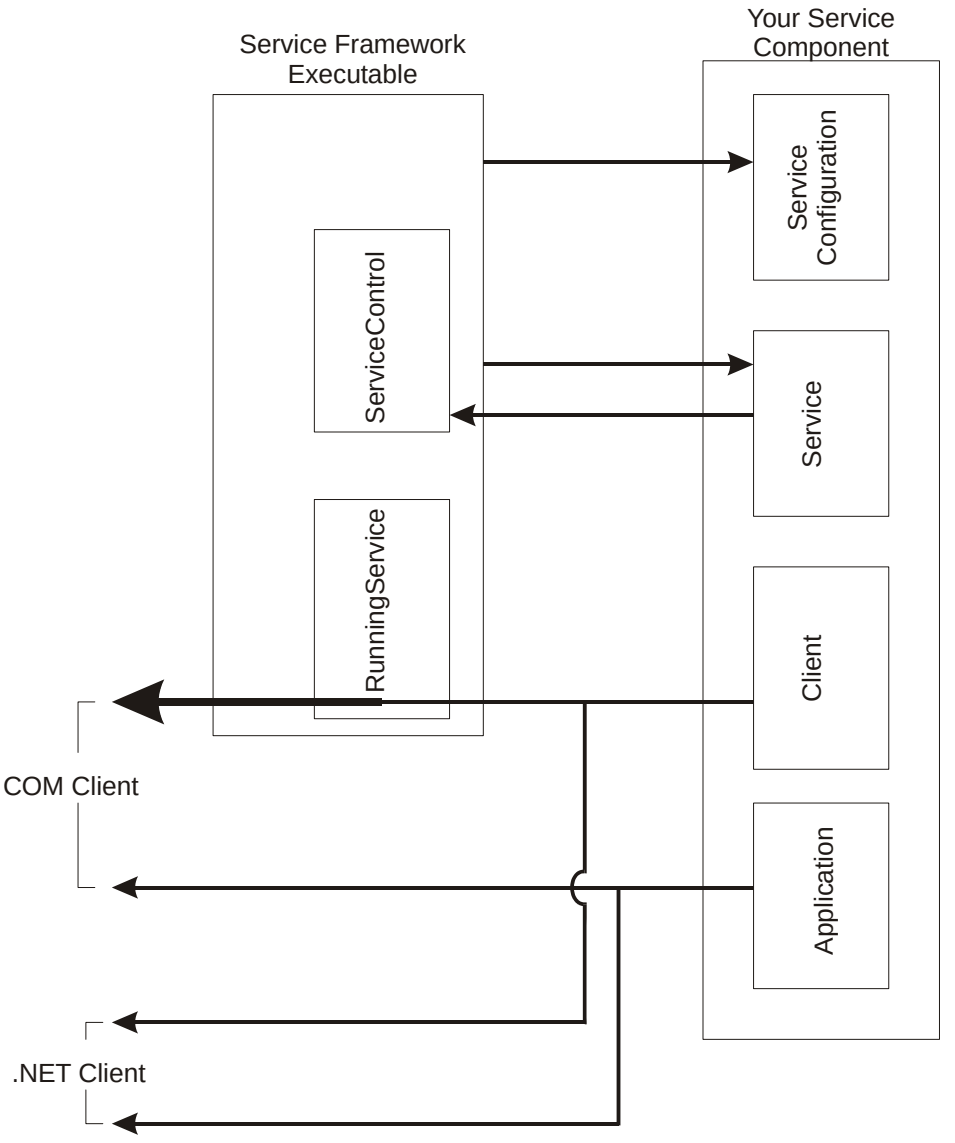

Figure 6
The Client Object Model

With a COM client, the RunningService object exposes all of the public methods and properties of your client object by reflecting them to the client through the RunningService object. You obtain a reference to the Application object by calling the GetAppObject method of the RunningService object.

With a .NET client, the actual client and application objects are referenced directly via .NET remoting.

Object Identifiers

In the COM edition of the toolkit, each of the IdwServiceClient interface methods included an object identifier value. This integer value made it possible to uniquely identify a client without holding a reference to the client (with the resulting circular reference problems). With .NET, you should use the references to the object itself to identify the object, as no circular reference problem exists. The ObjectIdentifier parameters have been removed from the interface methods.

The following functions are defined on the IdwServiceClient interface:

OnConnect

VB: OnConnect(ByVal ServiceObject As Object)

C#: OnConnect(Object ServiceObject)

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.

This call comes in immediately on the thread that creates the object, and is not synchronized.

OnDisconnect

VB: OnDisconnect(ByVal ServiceObject As Object)

C#: OnDisConnect(Object ServiceObject)

This method is called after a client has disconnected from the RunningService object.

This call is synchronized to the object being disconnected.

OnStop

VB: OnStop(ByVal ServiceObject As Object)

C#: OnStop(Object ServiceObject)

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

Note: *I*f you design your object such that it must receive this method call then it is your responsibility to defer the service from stopping until all of the client's OnStop methods are called.

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 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.

This method is called on the primary service thread. It is not, however, synchronized against other method calls on the object.

ExecuteBackground

If you call the ClientExecuteBackground method of the Service control object for a given client, 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. A new thread will be created and this method will be called by that thread.

Note that this method will not block other methods from being called (say, on additional client requests). You must perform your own synchronization to prevent threading problems.

The synchronization issue represents a major change from the COM edition of the toolkit. In the COM edition, the COM threading rules insured that during such background execution, no other calls into the object would occur. Under .NET, this protection does not exist.

To replicate the COM approach, you should add a SyncLock/lock statement to this method and any other methods that you wish to synchronize. If you use the client object itself as the synchronization object (with SyncLock/lock), you will automatically synchronize to the OnDisconnect method.

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 .NET, 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.

Shared Variables

Client objects should avoid accessing shared variables including global variables (VB) or static members of classes. Any time shared variables are accessed, you should be careful to synchronize that access to reduce the chance of threading problems.

The only shared variable that may be safely accessed is the service control object (IdwServiceCtl methods only), which are synchronized to the service control object (meaning only one thread can ever call them at a time).

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!

Pausing a service has no impact on the behavior of the client or application object. If you wish to change their behavior (say, respond with an error) when a service is paused, it is up to you to do so in your own code.

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 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:

4. Is a particular user allowed to perform a certain operation on a system?
5. 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 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.

Under DCOM, the client actually passes identity credentials to the server, making it possible for the client to specify who the user is and the type of impersonation that is permitted. In that case, 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!

Under .Net remoting, the default channels and configuration do not include identity credentials, so impersonation of a client is not possible unless you perform additional customization steps that are beyond the scope of this document.

However, 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 members of the System.Security.Principal.WindowsIdentity class. An example of this is shown in the RemoteUser example. Note, however, that unless you build a secure channel, the logon name and password in this remoting scenario is passed in plain-text, and is thus not secure.

Users migrating from the COM edition of the toolkit will note that the RemoteUser example has been modified to use LogonUser instead of client impersonation. This approach is supported under both DCOM and .NET remoting.

Configuring Your Service for 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 Remoting/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 executable on your service's computer, and access it via .NET remoting or 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 for .NET remoting is to install one of the simple console program based remoting examples provided with the .NET framework and make sure you can access the server from the client machine. If you do not know how to do this, refer to the .NET MSDN documentation and learn how to do this before you proceed to attempt this with a service.

The easiest way to make sure that your computers are configured properly for COM is to create a simple remotable ActiveX EXE server in VB6 that uses remote automation. Register its VBR file on the client machine 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 .NET Remoting

All objects in your service that are to be available via .NET remoting should be specified in your service configuration file as described earlier.

We strongly recommend Rammer's book on Advanced .Net Remoting or Advanced .Net Remoting with VB.Net to help you understand how to use .Net remoting.

Services created with this toolkit are true services, and are fully compatible with the .Net remoting system – so all the configuration options and framework classes related to remoting work just as you would expect.

The only circumstance where our toolkit provides additional remoting features are:

1. The ability to expose one object type simultaneously via .Net remoting and DCOM remoting.
2. The ability to give remoted objects access to service related information via the IdwServiceClient interface.

Configuring the Client System for Access to .NET Remoting Objects Exposed from the Service

A client can specify objects that should be accessed remotely by creating a configuration file which can be loaded using the following code:

```
System.Runtime.Remoting.RemotingConfiguration.Config
ure("ClientTest.exe.config")
```

The configuration file typically resembles the following:

```
The configuration file
<configuration>
  <system.runtime.remoting>
    <application name="ClientTest">

      <client
url="HTTP://localhost:8001/dwEasyServ">
        <activated type="dwEasyServ.Client,
dweasyservmetadata" />
      </client>
      <channels>
        <channel ref="http" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

The application name refers to your client application. The client url specifies the port and application name that was set by the server (in the server configuration file). The activated type specifies the type which will be accessed remotely, and the assembly that contains the type schema.

Once this file is successfully processed, all attempts to create the object dwEasyServ.Client (which is presumably exposed from your running service) will be redirected via remoting to the service.

The only remaining question is – how does the client obtain the type schema?

There are several different ways to accomplish this (all discussed in the .Net remoting documentation and Rammer's book). The one we recommend is to extract the metadata from your service using the soapsuds command as shown here:

```
soapsuds -types:dwEasyServ.Client,dwEasyServ -
oa:dweasyservmetadata.dll -nowp
```

This command extracts the metadata from the dwEasyServ.Client object from the service object, and places it in an assembly named dweasyservmetadata.dll, which can be referenced by the client application to obtain the schema for the type.

Users of the COM version of the toolkit will notice that the RemoteUser sample application, which in the past returned an ADO recordset, now returns a string array. It is possible to remote an ADO recordset via .Net remoting, but it is complex and requires customization of the metadata file to avoid duplicate references to the ADODB namespace.

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 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 resolve 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.

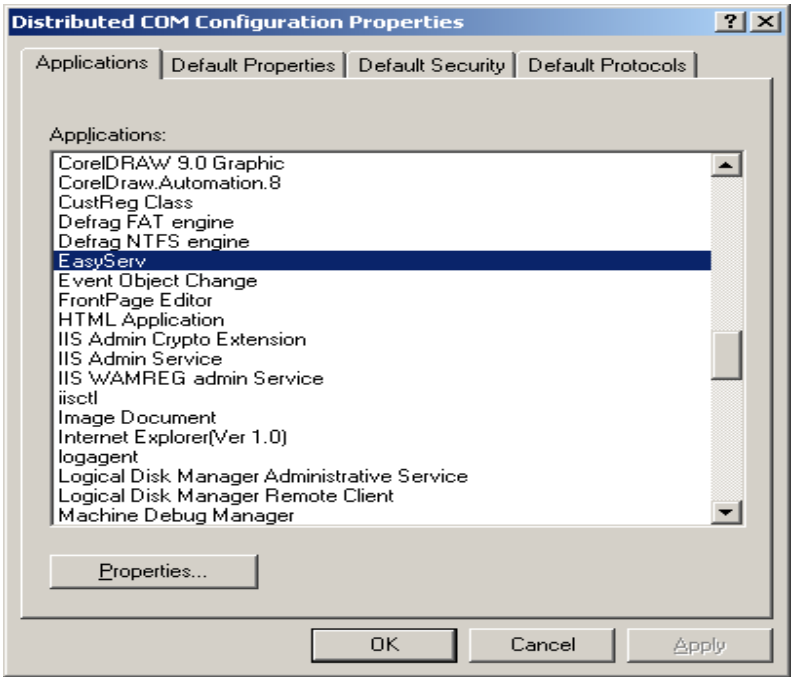

Figure 7

Distributed COM Configuration Properties

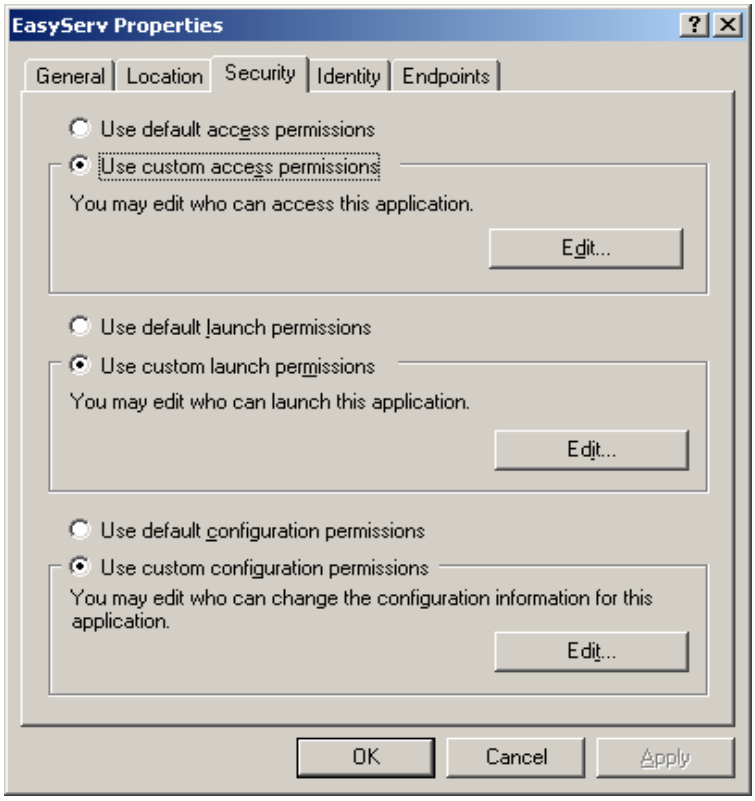

Figure 8
EasyServ Properties

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:

1. The name and class identifier (CLSID) of the object, and other information needed to register the object.

2. 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).
3. 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. You need to generate a new VBR file each time you recompile your Service EXE.

Copy the VBR file to the client system. Then run the clireg32 program to register the VBR file. The CliReg32 program will prompt you for the IP address or name of the machine.

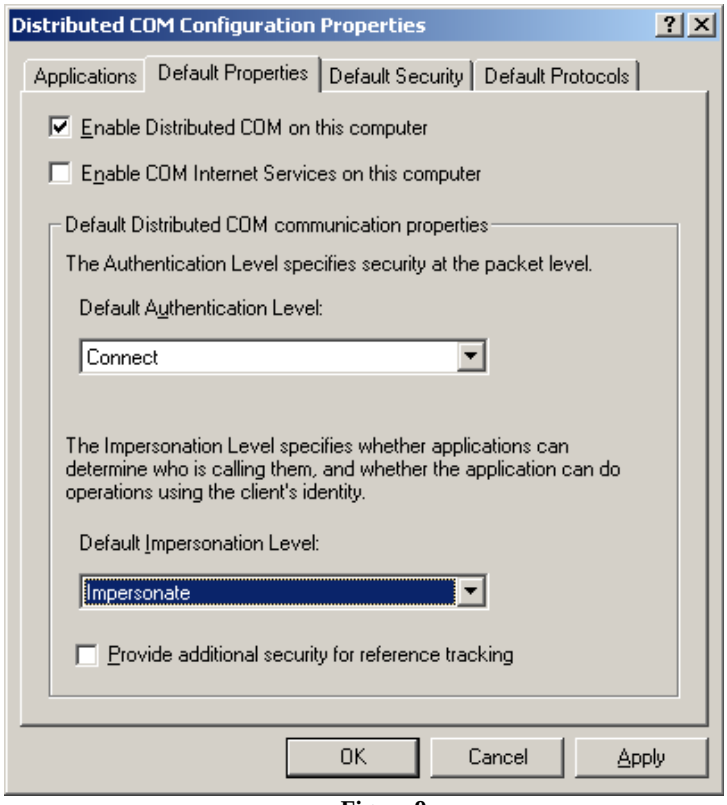

Figure 9
Distributed COM Configuration Properties

Examples

This 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 that may arise while testing and debugging the example.

All of the examples for the demo edition have the same assembly name (dwEasyServ) 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.
ControlPanelApplet	Sample shows how to create a standalone control panel applet.
ControlPanelAppletService	Sample shows how to create a control panel applet that interacts with your service.
ReportEvent2	Similar to the Beeper sample. Demonstrates how to use the ReportEvent2 function to perform advanced event logging operations.
Tracing	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.
TinyWeb2	This slightly less primitive web server returns the same page using the HTTP protocol. Demonstrates use of threading to service HTTP requests on multiple threads, thus preventing connections from blocking each other.
FileWatch	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. MSMQ can also be used to allow applications to send messages to a service. This example requires that MSMQ be installed to run.
FileWatch2	Similar to the FileWatch sample except this does not require MSMQ. Changes are output to the Tracing log file or the Event Log.
RemoteUser	This service allows users to access a database record set and a file using .NET remoting, COM or DCOM. • Demonstrates .NET remoting and 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 impersonation – acting in the security context of the client. • Demonstrates holding a service open until all clients disconnect.
StateCoder	This service demonstrates how to implement state machines using Desaware's StateCoder in your service.
COM Service Library	The COM edition of the Service Control Manager component. Can be used to start and stop services, and retrieve information on running services. This edition wraps the Windows Service API functions and exposes almost all of the Services capabilities. Includes full VB6 source code. NOTE: This folder is located as a subfolder of the dwSCMDemonstrationCOM folder and is installed only when the *Install Visual Basic .NET files* option is selected.
dwSCMNet	A managed .NET edition of the Service Control Manager component. Can be used to start and stop services, and retrieve information on running services. This edition wraps the native .NET Service objects and exposes just a subset of the Services capabilities. Includes full VB.NET source code.
DwSCMDemonstration-Net	This sample demonstrates how to use the .NET edition of the Service Control Manager component.
DwSCMDemonstration-COM	This sample demonstrates how to use the COM edition of the Service Control Manager component in a .NET application.

Migration FAQ

Migration Issues Relating to the Transition From the COM Edition Toolkit

The following are common questions regarding the migration from the COM edition of the toolkit to the .NET edition of the toolkit.

Where is the dwSecurity Object?

The dwSecurity object and DLL was designed to provide an in-process component for performing various security related tasks associated with impersonation. These tasks can be accomplished using classes and methods from the System.Security.Principal namespace.

Where is the dwBackThread Object?

The dwBackThread Object allows VB6 programs and components to create and manage background threads. Multithreading is well supported in .NET (though it is certainly as tricky as with VB6, and considerably more risky). Refer to the objects of the System.Threading namespace for specifics on multithreading. Refer to Dan Appleman's book "Moving to VB.Net: Strategies, Concepts and Code" for assistance with learning how to do multithreading safely with .NET.

Where is the dwSock Component?

The dwSock component provides a powerful winsock wrapper for VB6 programmers. It is no longer necessary with the .NET framework. Refer to the classes and objects in the System.Net and System.Net.Sockets class to implement network access from your services.

NOTE: The dwSock6.dll file is still distributed with the NT Service Toolkit .NET edition but we are not supporting the dwSock6 component in .NET services. From our preliminary tests, it seems to work fine in .NET, but it has not undergone the rigorous testing in .NET we normally apply to our native .NET components. For those who still wish to use the dwSock6 component in your .NET projects, an example of the TinyWeb project that uses the dwSock6 component is included in the sub folder of TinyWeb.

Migration Issues Relating to the Transition from VB6

How can the interface names be the same in the .NET edition, even though the interfaces are different?

This is possible because the .NET interfaces are not COM interfaces, nor are they really the same name. For example: The full name of the IdwEasyService interface is actually:

Desaware.ServiceToolkit.IdwEasyService.

In .NET, the full namespace is what counts.

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 installed in the same folder as the Service EXE. Also verify that the Desaware.ServiceToolkit.Interfaces.dll file is distributed with your service files and installed in the same folder as your service files or in the GAC.

Client Objects

Permission Denied Error When Creating the RunningService Object from COM/DCOM

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.

Unable to Access an Object Through .NET Remoting

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

- Be sure your service is configured to use .NET remoting. The framework will attempt to load a remoting configuration file named yourdll.config, where yourdll is the name of the service assembly DLL without the .DLL extension. The remoting configuration file should be installed in the same folder as your service assembly DLL file.
- Be sure your clients are initializing the remoting infrastructure correctly. The samples included demonstrate the use of the RemotingConfiguration object's Configure function to read the configuration file to initialize the remoting infrastructure. There are additional methods you can use to initialize the remoting infrastructure.
- Be sure your client is configured correctly to access the remote service. Channel type, port number, and object names must be correct.
- If you are using port numbers, be sure another service is not using the same port number. If the same port number is used a conflict may occur. To resolve this conflict, you must first stop the service that is causing the conflict and then restart your service .

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 the service is running as a service with a compiled VB component.

- Be sure the VBR file is registered on the client system using the clireg32 application.
- Be sure that the correct VBR file is installed on the client system. You must generate a new VBR file each time you recompile your Service EXE.

While Running

The Service Stops Working

It is critical that your service component not raise unhandled runtime errors. These errors will be caught by the framework and will cause the service to stop. Use the tracing and logging capability to detect when this has occurred. Setting a background "beep" on the primary 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 the Desaware NT Service Toolkit than it is to debug traditional services written using Visual C++! This is because the simulator mode allows you to test and debug most aspects of your service without actually installing it as a service.

Tracing and Logging

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

You can initiate logging by setting the TraceLevel switch in your service's configuration file to a number between 1 and 4. The service configuration file is an XML file with the same name as the service plus the extension .config. Thus, if your service name is Beeper.exe, the configuration file name will be Beeper.exe.config. The configuration file should be in the same directory as the service.

The following is a typical configuration file that sets the TraceLevel switch to 1 (severe errors).

```
<configuration>
    <system.diagnostics>
       <switches>
          <add name="TraceLevel" value="1" />
       </switches>
    </system.diagnostics>
</configuration>
```

The TraceLevel setting determines what information is written to the log file. When the TraceLevel entry is zero, or not present, 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.

You can send tracing output to a log file or any other destination by setting an entry in the Listeners field. For example:

```
<configuration>
    <system.diagnostics>
       <switches>
```

```
            <add name="TraceLevel" value="4" />
         </switches>
         <trace autoflush="true" indentsize="4">
            <listeners>
               <add name="myTracingNET_VBListener"

type="System.Diagnostics.TextWriterTraceListener,Sys
tem,Version=1.0.3300.0,Culture=neutral,PublicKeyToke
n=b77a5c561934e089"
                initializeData="c:\errors.log" />
               <remove name="Default"/>
            </listeners>
         </trace>
      </system.diagnostics>
</configuration>
```

Sends trace information to the file c:\errors.log. **Note:** You should specify a path for the output log file. Otherwise, the file will be written to the default trace listener path (which will probably not be the same path where your service exe resides) which may vary.

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.

For more information on tracing in .NET, refer to Dan Appleman's eBook "Tracing and Logging with .NET". For details visit http://www.desaware.com/tracing.htm.

Testing and Debugging – Simulator Mode

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

- Build your service executable using the Service Configuration Wizard.
- 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).

- Open your service object project in Visual Studio.
- Set the debug project settings to launch your service executable, with the extension –Sim.
- Run your project in Visual Studio.

The simulator window will appear that allows you to pause, continue, stop and shutdown your service. You can set breakpoints in your component and verify the operation of the component using the Visual Studio .NET environment.

***Note**: Be sure your debug Exceptions options are set to "Break into debugger". If you have it set to "continues", the service framework will receive the error and interpret it as a failure in your component and terminate the service.*

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 user currently logged on. You must verify operation of your service while running as a service

To debug a running service, do the following:

- 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.
- Select the Debug-Processes dialog and attach the Visual Studio debugger to the running service. You may need to check the "Show System Processes" checkbox to view the service.

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

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

When running under the simulator, your service is running in the current interactive user account. It is critical that you test your service as a service, while running under the account that will be used by the service itself.

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. Please 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.

The 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 .NET. There are two versions of this library. The original version (dwSCM.dll) is a COM component written and compiled in Visual Basic 6 that provides a full set of features. The second (dwSCMNet.dll) is a version built using only the .NET framework and .NET class objects, but it has fewer features.

These are the changes in the .NET framework version of the library:

dwServiceManager Object

LockServiceDatabase, UnlockLockServiceDatabase, QueryLock-Status, GetServiceNameFromDisplayName, GetDisplayNameFrom-ServiceName and CreateService are not implemented. Initialize-SCManager now always returns zero (as there are no handles in the .NET framework service functions) and the database parameter is not used.

dwServiceObject Object

ServiceHandle, ChangeServiceConfig, and DeleteService are not implemented. There is a new read only property called **Service** of type ServiceProcess.ServiceController, for retrieving the ServiceController object.

dwServiceStatus Object

Win32ExitCode, ServiceSpecificExitCode, CheckPoint and WaitHint are not implemented.

dwServiceConfig Object

Only ServiceType, ServiceName, DisplayName and Dependencies are implemented, and they are read-only.

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 used by the service. 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

COM

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

.NET

InitializeSCManager
(ByVal Machine As String, ByVal Database As String, ByVal Access As ServiceControlRights) As Integer

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. This return value is not needed for any functions in the dwSCM library. **No value is returned in the .NET version.**

```
Dim sc as New dwServiceManager
Call sc.InitializeSCManager ("","",
ServiceControlRights.SC_MANAGER_ALL_ACCESS)
```

COM

EnumServicesStatus
(Optional ByVal ServiceType As Long = SERVICE_WIN32, Optional ByVal ServiceState As

EnumServiceStates = SERVICE_STATE_ALL) As Collection

.NET

EnumServicesStatus

(Optional ByVal ServiceType As Integer = 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.

VB6

```
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
```

VB.NET

```
Dim svcstat As dwServiceStatus
Dim ServiceCollection as Collection

ServiceCollection = sc.EnumServicesStatus()
' Now the names and descriptions are printed.
For Each svcstat In ServiceCollection
    Debug.WriteLine svcstat.Name
    Debug.WriteLine svcstat.DisplayName
```

```
Next
```

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

The OpenService method 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. In the .NET edition, an exception is raised if the specified Service is not found or could not be opened.

VB6

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject
' Open the service to change its configuration.
Set servobj = sc.OpenService (servname,_dispname,
SERVICE_CHANGE_CONFIG)
```

VB.NET

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject

servobj = sc.OpenService (servname,_dispname,
ServiceAccessRights.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. **COM edition only**.

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. **COM edition only**.

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. **COM edition only**.

UnlockLockServiceDatabase () As Boolean

If you have used LockServiceDatabase method, be sure to call this immediately after you have completed modifying a service. **COM edition only**.

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 regarding who created a lock. FALSE is returned if the SCM is not currently locked. **COM edition only**.

```
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 determine when the service has responded. An error is raised if the service cannot start.

VB6

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject

Set servobj = sc.OpenService (servname, dispname,
SERVICE_START)

On Error GoTo FailStart
servobj.StartService
```

VB.NET

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject

servobj = sc.OpenService (servname,_dispname,
ServiceAccessRights.SERVICE_START)

Try
    servobj.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. You can also send numbers with values up to 255 to notify the service to perform a custom function (as long as the service is not stopped or paused).

VB6

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject

Set servobj = sc.OpenService (servname, dispname,
SERVICE_STOP)
servobj.ControlService SERVICE_CONTROL_STOP
```

VB.NET

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject

servobj = sc.OpenService (servname, dispname,
SERVICE_STOP)
servobj.ControlService SERVICE_CONTROL_STOP
```

QueryServiceStatus() As dwServiceStatus

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

VB6

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject
Dim ss As dwServiceStatus

Set servobj = sc.OpenService (servname, dispname,
SERVICE_QUERY_STATUS)

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

VB.NET

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject
Dim ss As dwServiceStatus

servobj = sc.OpenService (servname, dispname,
SERVICE_QUERY_STATUS)
ss = OpenService.QueryServiceStatus()
Debug.WriteLine (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.

VB6

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject
```

```
Dim scfg As dwServiceConfig

Set servobj = sc.OpenService (servname, dispname,
SERVICE_QUERY_CONFIG)
Set scfg = servobj.QueryServiceConfig()
Debug.Print sc.AccountName
```

VB.NET

```
Dim sc As New dwServiceManager
Dim servobj as dwServiceObject
Dim scfg As dwServiceConfig

servobj = sc.OpenService (servname, dispname,
SERVICE_QUERY_CONFIG)
scfg = servobj.QueryServiceConfig()
Debug.WriteLine 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. **COM edition only**.

```
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)
```

COM

EnumDependentServices
(ByVal ServiceState As Long) As Collection

.NET

EnumDependentServices
(ByVal ServiceState As Integer) As Collection

If you need to stop a service, you should also stop all services that depend upon it – otherwise the dependent services might fail or even cause exceptions. EnumDependentServices returns a collection of dwServiceStatus objects representing all the services that require the original 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 computer is shutdown and restarted. This does not effect the service files. **COM edition only**.

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. **COM edition only**.

Service As ServiceProcess.ServiceController

This is a read only property that returns the .NET framework service object that represents the dwServiceObject service. **.NET edition only**.

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. **COM edition only**.

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. **COM edition only**.

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. **COM edition only**.

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. **COM edition only**.

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. **COM edition only**.

ErrorControl as ServiceErrorControlType

ErrorControl is a value of the ServiceErrorControlType enumeration. If the service fails to start, this will determine how seriously the system consider the situation, and how the system will respond. If you are creating a dwServiceConfig instance for use with Change-ServiceConfig and you do not want to change this value, set it equal to SERVICE_NO_CHANGE. **COM edition only**.

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. **COM edition only**.

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. **COM edition only**.

TagId As Long

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

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 under which the service is logged.

If you are creating a dwServiceConfig instance for use with Change-ServiceConfig and you do not want to change this value, do not modify this property. **COM edition only**

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. **COM edition only**

DisplayName as String

DisplayName is a user-friendly description of the service's purpose. If you are creating a dwServiceConfig instance for use with Change-ServiceConfig 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 LockService-Database and UnlockSer-viceDatabase functions.
SC_MANAGER_QUERY_LOCK_STATUS	Enables the Query-Service LockStatus 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 EnumDependent-Services function.
SERVICE_START	Enables StartService.
SERVICE_STOP	Enables calling the Control-Service 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_SHUTD OWN	f this value is set, the service will be notified of when the system is shutting down.
SERVICE_ACCEPT_PARAM-CHANGE	If this value is set, the service can accept new startup parameters while it is running. This is only valid in Windows 2000/XP.

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.

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 .NET class library project (or modify the template file provided). Perform the modifications listed later in this section.
3. Test the control panel applet by copying your class library into the primary system folder (System32). Place your CPL file in the System32 folder.

   NOTE: If you are distributing your control panel applet, be sure that you also distribute the Desaware.ServiceToolkit.Interfaces.dll file. This file needs to be installed into the GAC – a merge module is included for this purpose.

   NOTE: In Windows 2000 and later, you can install your Control Panel Applet into a folder other than the System folder. If you do so, you must add a string value to the following HKEY_ LOCAL_MACHINE registry key - Software\Microsoft\Windows\ CurrentVersion\Control Panel\Cpls in order for the system Control Panel to be able to include your control panel applet in its list. The name of the string value can be a description for your control panel applet, the string value must contain the full path and name of your control panel applet 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 Studio 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.

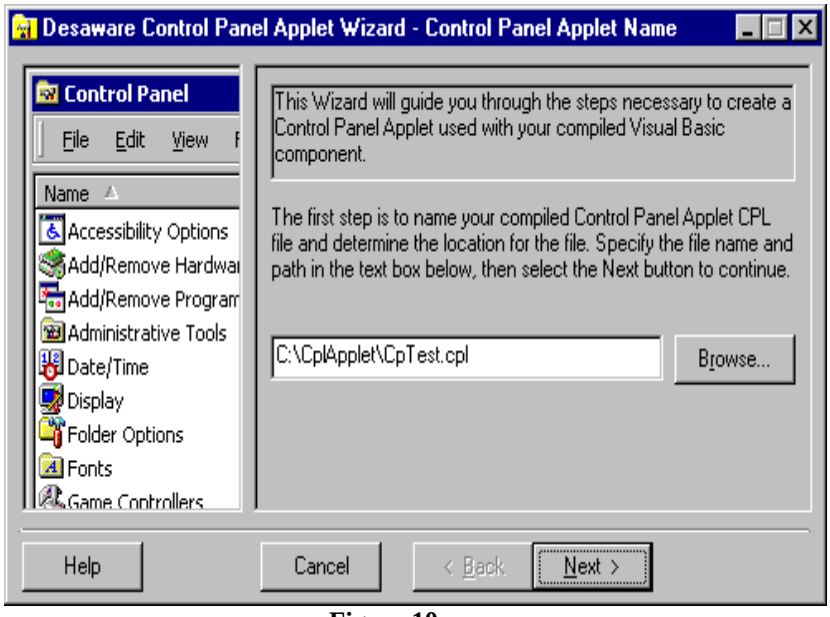

Figure 10
Control Panel Applet Wizard

Assembly Name

Enter the assembly name of the .NET 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 assembly name, you will have to run this configuration program again.

It is important that you should choose an assembly name that is unique – duplication of assembly 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 assembly name length can be up to 32 characters.

Version Information

Enter the version resource information for your service executable file. The control panel applet Wizard writes a standard version resource into your control panel applet. 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 template class file.

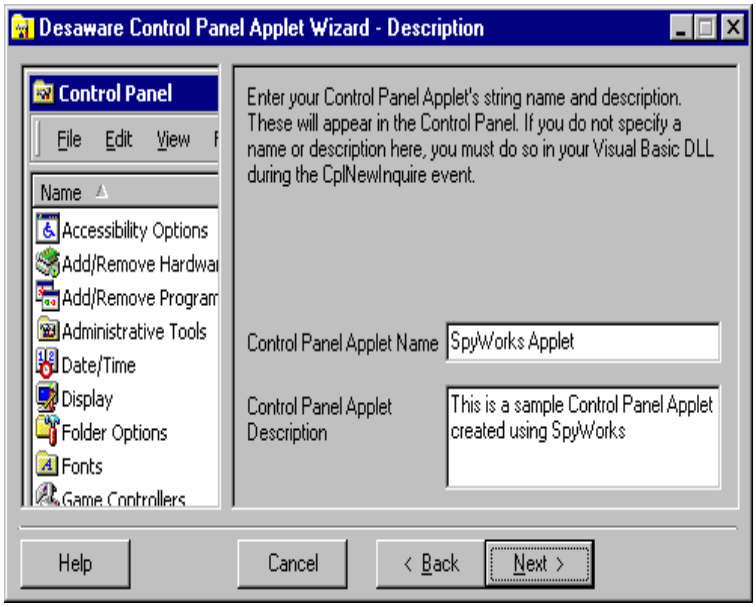

Figure 11
Control Panel Applet Wizard – Description Dialog Box

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 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 or not the compilation was successful. 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 (refer to the earlier section *Building a Control Panel Applet* for registry requirements).

System Compatibility

The Control Panel Applet Wizard requires Windows 2000/XP to run.

Create an Assembly DLL for Your Control Panel Applet

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

VB:

```
Imports Desaware.ServiceToolkit
Public Class Applet
    Implements IcontrolPanelApplet
```

C#:

```
using Desaware.ServiceToolkit;
public class Applet: IControlPanelApplet
```

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.

CplDblClk

(ByVal AppNumber As Integer, ByVal UserData As Integer)

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:

VB:

```
Public Sub CplDblClk(ByVal AppNumber As Integer, _
ByVal UserData As Integer) Implements _
IControlPanelApplet.CplDblClk

   Dim frm As New frmApplet()
   frm.ShowDialog()
```

C#:

```
void IControlPanelApplet.CplDblClk(int AppNumber,
int UserData)

   frmApplet frm = new frmApplet();
   frm.ShowDialog();
```

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.

VB:

```
Public Sub CplExit() Implements _
IControlPanelApplet.CplExit
```

C#

```
void IControlPanelApplet.CplExit()
```

CplGetCount() As Integer

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:

VB:

```
Public Function CplGetCount() As Integer Implements
IControlPanelApplet.CplGetCount
   Return (1)
```

C#:

```
int IControlPanelApplet.CplGetCount()
   return (1);
```

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

CplInit() As Integer

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.

VB:

```
Public Function CplInit() As Integer Implements _
IControlPanelApplet.CplInit
   Return (1)
```

C#:

```
int IControlPanelApplet.CplInit()
   return (1);
```

CplInquire

(ByVal AppNumber As Integer, ByRef idIcon As Integer, ByRef idName As Integer, ByRef idInfo As Integer, ByRef UserData As Integer)

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 CplDblClick method.

If the AppNumber value is 0, the idIcon, 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 idIcon, idName and idInfo must be left at the default value (-1) for all applets other than the first one.

VB:

```
Public Sub CplInquire(ByVal AppNumber As Integer, _
ByRef idIcon As Integer, ByRef idName As Integer, _
 ByRef idInfo As Integer, ByRef UserData As _
 Integer)  Implements IControlPanelApplet.CplInquire
```

C#:

```
void IControlPanelApplet.CplInquire(int AppNumber,
ref int idIcon, ref int idName, ref int idInfo, ref
int UserData)
```

CplNewInquire

(ByVal AppNumber As Integer, ByRef hIcon As Integer, ByRef szName As String, ByRef szInfo As String, ByRef UserData As Integer)

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 CplDblClick method.

VB:

```
Public Sub CplNewInquire(ByVal AppNumber As _
Integer, ByRef hIcon As Integer, ByRef szName _
As String, ByRef szInfo As String, ByRef UserData _
As Integer) Implements _
 IControlPanelApplet.CplNewInquire
```

C#:

```
void IControlPanelApplet.CplNewInquire(int
AppNumber, ref int hIcon, ref string szName, ref
string szInfo, ref int UserData)
```

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 Integer, ByVal ExtraData As String) As Boolean

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

VB:

```
Public Function CplStartWParms(ByVal AppNumber _
As Integer, ByVal ExtraData As String) As _
Boolean Implements _
 IControlPanelApplet.CplStartWParms
```

C#:

```
bool IControlPanelApplet.CplStartWParms(int
AppNumber, string ExtraData)
```

CplStop

(ByVal AppNumber As Integer, ByVal UserData As Integer)

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.

VB:

```
Public Function CplStop(ByVal AppNumber As _
Integer, ByVal UserData As Integer) As _
Integer Implements IControlPanelApplet.CplStop
```

C#:

```
int IControlPanelApplet.CplStop(int AppNumber, int
UserData)
```

Using Control Panel Applets with Services

The dwCPLService example provided with the toolkit illustrates how you can communicate with your service using the control panel applet. The sample project references the dwSCMNet component to access the Service functionality. The InitializeSCManager function is used to initialize the service control manager. The OpenService function then opens the specified service and returns a dwServiceObject object. The ControlService function can be used to pass values to a running service – calling this function ultimately leads to your services IdwEasyService_ OnUserControlCode method being called.

A partial listing of the relevant code follows:

VB.NET:

```
Imports dwSCMNet

Const USERIDFORSHORTESTTIMEOUTVALUE As Short = 128
Const USERIDFORSHORTERTIMEOUTVALUE As Short = 129

Dim scm As New dwServiceManager()
Dim hservice As dwServiceObject

Private Sub frmServiceApplet_Load(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
    Dim sStatus As dwServiceStatus

    hservice = Nothing
    Call scm.InitializeSCManager("", "",
ServiceControlRights.SC_MANAGER_CONNECT)

    Try
        hservice = scm.OpenService("dwBeeperNET_VB",
"", ServiceAccessRights.SERVICE_INTERROGATE Or
ServiceAccessRights.SERVICE_USER_DEFINED_CONTROL)
    Catch exc As Exception
        ' Service not found, hide the buttons
        btn250ms.Visible = False
        btn1500ms.Visible = False
        Exit Sub
    End Try

    If hservice.QueryServiceStatus.CurrentState <>
ServiceStateConstants.SERVICE_RUNNING Then
        ' Service is installed but not currently
running, hide buttons and display error message.
```

```
        lblWarning.Text = "Service '" +
hservice.ServiceName + "' not Running"
        btn250ms.Visible = False
        btn1500ms.Visible = False
    Else
        lblWarning.Visible = False
    End If

End Sub

' btn1500ms and btn250ms are simple commands.
' In this case they set the beep duration to fixed
values.

Private Sub btn250ms_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
btn250ms.Click
    Try

hservice.ControlService(CType(USERIDFORSHORTESTTIMEO
UTVALUE, ServiceControlConstants))
    Catch exc As Exception
        lblWarning.Text = exc.ToString
        lblWarning.Visible = True
    End Try

End Sub

Private Sub btn1500ms_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
btn1500ms.Click
    Try

hservice.ControlService(CType(USERIDFORSHORTERTIMEOU
TVALUE, ServiceControlConstants))
    Catch exc As Exception
        lblWarning.Text = exc.ToString
        lblWarning.Visible = True
    End Try
    End Sub
```

C#:

```
using dwSCMNet;

short USERIDFORSHORTESTTIMEOUTVALUE = 128;
short USERIDFORSHORTERTIMEOUTVALUE = 129;

dwServiceManager scm;
dwServiceObject hservice;

private void frmServiceApplet_Load(object sender,
System.EventArgs e)
```

```
{
    dwServiceStatus sStatus;

    scm = new dwServiceManager();

    hservice = null;
    scm.InitializeSCManager("", "",
ServiceControlRights.SC_MANAGER_CONNECT);

    try
    {
        hservice = OpenService("dwBeeperNET_C", "",
ServiceAccessRights.SERVICE_INTERROGATE |
ServiceAccessRights.SERVICE_USER_DEFINED_CONTROL);
    }
    catch (Exception exc)
    { // Service not found, hide the buttons
        btn250ms.Visible = false;
        btn1500ms.Visible = false;
        return;
    }

    if (hservice.QueryServiceStatus().CurrentState
!=.ServiceStateConstants.SERVICE_RUNNING)
    {
        // Service is installed but not currently
running, hide buttons and display error message.
        lblWarning.Text = "Service '" +
hservice.ServiceName + "' not Running";
        btn250ms.Visible = false;
        btn1500ms.Visible = false;
    }
    else
    {
        lblWarning.Visible = false;
    }

}


// btn1500ms and btn250ms are simple commands.
// In this case they set the beep duration to fixed
values.
private void btn250ms_Click(object sender,
System.EventArgs e)
{
    try
    {

hservice.ControlService((ServiceControlConstants)USE
RIDFORSHORTESTTIMEOUTVALUE);
    }
    catch (Exception exc)
```

```
    {
        lblWarning.Text = exc.ToString();
        lblWarning.Visible = true;
    }
}

private void btn1500ms_Click(object sender,
System.EventArgs e)
{
    try
    {

hservice.ControlService((ServiceControlConstants)USE
RIDFORSHORTERTIMEOUTVALUE);
    }
    catch (Exception exc)
    {
        lblWarning.Text = exc.ToString();
        lblWarning.Visible = true;
    }
}
```

Installing and Testing Your Control Panel Applet

It is very important that your control panel applet DLL and the Desaware.ServiceToolkit.Interfaces??.dll (where ?? is 11 or 20 depending on .NET framework version) file be available on the system when your control panel applet CPL file is installed. The Desaware.ServiceToolkit.Interfaces??.dll file should be installed into the GAC, and the control panel applet DLL and CPL files should be installed in the same folder.

When testing your control panel applet, you may want to initially avoid opening the Control Panel. This is because once you open the Control Panel, it maps your control panel applet DLL and CPL files to the Explorer.exe process space. If you want to recompile either of these files, you will get an access denied error and you will need to log off before the Explorer process releases these modules. You can test your control panel applet by selecting your control panel applet CPL file name with Windows Explorer, then right clicking on the file, then selecting the “Open With Control Panel” menu item. This method does not permanently map your control panel applet files into Explorer.exe.

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

1. Copy all required files into the system directory (the only way to install it under versions of Windows before Windows 2000/XP).

2. In Windows 2000/XP, copy all required files into the directory of your choice, then add the CPL file to the registry as shown in the next section.

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
```

```
MyCpl="%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 .NET control panel applet assembly DLL file.	This is the .NET assembly file you create to implement the control panel applet.
Desaware.ServiceToolkit.Interfaces??.dll	This is the .NET component file used internally by the framework to marshal control panel applet interfaces. Install this file in the GAC.

Installing and Distributing Your Service

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

In addition to all of the usual issues involved in distributing a .NET component (making sure you ship all the necessary dependent components and runtimes, and installing and registering 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. 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 Desaware.ServiceToolkit.Interfaces.dll file 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 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.
- 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 *paramstring*] [-Silent]	Register your service to run in standalone mode with the simulator. /Params *paramstring* – 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 spaces.
-UnRegserver [-Silent]	Unregister your service from standalone mode.
-I [-User *user* -Password	Install your service to run as a service. -User *username* – If specified, this overrides the

password] [-Params *paramstring*]	user name provided by the service configuration file (if any).
[-Silent]	-Password *password* – 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 *paramstring* – 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 and the reinstall the service in order for those changes to take effect.
-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.

Redistributable Components

The following components can be distributed with your service.

- The service executable you create using the Service Configuration Wizard Program.
- The config template file created for you by the Service Configuration Wizard Program.

- The VBR file created for you by the Service Configuration Wizard Program.
- The Desaware.ServiceToolkit.Interfaces??.dll file (must be installed in the GAC).
- The control panel applet CPL file created using the Control Panel Applet Wizard.
- The dwSCMNet.dll (managed .NET edition) component for use with services.
- The dwSCM.dll (COM edition, must be registered) component for use with services.

If you use an installation program to prepare your service for distribution, (including the Windows installer), it may detect that your service is dependent on the file Desaware.StateCoder.dll even if you are not using the StateCoder integration features. Despite what your installation program says, you do NOT need to distribute the Desaware.StateCoder.Dll file if your service does not actually use StateCoder.

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 .NET remoting, COM/DCOM and security.

Desaware cannot provide technical support on the issue of authentication of users or basic .NET remoting, 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 .NET 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.

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) 377-4770 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) 377-4770 or email support@desaware.com.

msdn.microsoft.com

MSDN contain a wealth of information and sample code, plus the latest Visual Basic knowledge base.

Index

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 COM Edition, .NET Edition

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 (gàl´e-mô¹frê) noun
plural gal·li·mau·fries
A jumble; a hodgepodge.

[French galimafrée, from Old French galimafree, sauce, ragout : probably galer, to make merry. See GALLANT + mafrer, to gorge oneself (from Middle Dutch moffelen, to open one's mouth wide, of imitative origin).]
(From The American Heritage® Dictionary of the English Language, Third Edition copyright © 1992 by Houghton Mifflin Company)

What does a Twain control, spiral art program, set of linked list classes, a quick sort routine, a hex editor and a myriad of other custom controls have in
common?

They are all part of Desaware's ActiveX Gallimaufry.

You'll find most of these controls useful, the rest entertaining – but we guarantee that you'll find them all educational, because they come with complete Visual Basic 6.0 source code.

Curious?

Want to learn some advanced API programming techniques? Visit our web site for a full description and demo.

THE CUSTOM CONTROL FACTORY V 4.0
The Custom Control Factory is a powerful tool for creating your own animated buttons, multiple state buttons, toolbars and enhanced button style controls in Visual Basic and other OLE control clients, without programming. With 256 & 24 bit color support, automatic 3D backgrounds, image compression, over 50 sample controls and more. Plus MList2 - an enhanced listbox control. 16 & 32 bit ActiveX controls and 16 bit VBXs included.