



realtimepublishers.com™

The Definitive Guide™ To

Windows Desktop Administration

SCRIPTLOGIC

Bob Kelly

Chapter 7: Scripting Custom Solutions.....	159
Should You Script It?.....	159
Choosing a Scripting Language	162
VBScript	162
KiXtart	163
JScript	163
WinBatch	164
PerlScript.....	164
Shell Scripting.....	164
COM Automation	165
WMI.....	165
Active Directory Service Interfaces.....	165
WSH.....	165
ActiveX Data Objects	166
Internet Explorer	167
Automate Actions	167
Display Information.....	167
Running Your Scripts	169
Logon Scripts	169
RunOnce and Run.....	171
Startup Shortcuts.....	171
Schedulers	172
Manual Execution.....	173
Group Policy	174
Assigning Scripts	174
Building Lists.....	175
Machine List	176
Custom Script Examples.....	178
Restarting Remote Systems	179
Backing Up Event Logs	179
Software Inventory.....	180
The AutoLogon Process.....	182
Summary	184

Copyright Statement

© 2003 Realtimedpublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimedpublishers.com, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtimedpublishers.com, Inc or its web site sponsors. In no event shall Realtimedpublishers.com, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtimedpublishers.com and the Realtimedpublishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.


If you have any questions about these terms, or if you would like information about licensing materials from Realtimedpublishers.com, please contact us via e-mail at info@realtimedpublishers.com.

Chapter 7: Scripting Custom Solutions

Even the most robust, all-encompassing off-the-shelf solution cannot provide an answer for every problem you are likely to face. The more complex the requirements of your organization, the quicker you will encounter a need to develop your own solutions—perhaps by extending the capabilities of a management system or utility that you already own. Acknowledging this need, many management solution vendors include in their tools an ability to facilitate custom scripting by providing integration through Common Object Model (COM) automation, application programming interfaces (APIs), and even their own scripting languages.

Scripting and automation are synonymous. If you need to perform the same actions every day, or hundreds of times at once, the need for a utility or script becomes clear. Many command-line tools offer functions designed for use in scripts. For example, suppose you want to reboot several computers; you can use the shutdown.exe utility that has been around for years, but it has a limitation—it accepts only a single computer name as a parameter. Although you no longer have to visit hundreds of machines to reboot them, you will still have to type each individual computer name hundreds of times as a parameter for this utility. However, you can write a script that can read your list of computers and execute the command line for each one. Doing so lets you spend your time writing a script instead of writing out a batch file or typing the shutdown command over and over again:

```
Shutdown.exe -r -m \\PC7463
Shutdown.exe -r -m \\PC3772
Shutdown.exe -r -m \\PC9283
Shutdown.exe -r -m \\PC0139
```

 Later in this chapter, we will cover a script that you can use to restart multiple machines.

Should You Script It?

If it is faster to script a task than it is to perform the task manually, the decision of whether to script the task seems like an obvious decision. However, if you will need a week to script a task or a week to perform the task manually, what then? Consider the following benefits of scripting:

- **Reuse**—To use the previously mentioned shutdown script as an example, the decision to script the task might be questionable if only 50 machines need to be restarted. However, you must consider the fact that you will have this script available the next time those machines need to be restarted. If restarting computers is something you do often (as is usually the case for systems administrators who use Group Policy for software distribution), a script that does the job can be quite valuable.
- **Customization**—The ability to customize every aspect of your solution is a major benefit of scripting. However, this benefit comes with a drawback: Once you start to produce custom solutions, you will soon be asked for further customization and scripting can quickly become a large part of your day. (Tasks management might have originally seen as technically unfeasible will suddenly start to seem within their grasp—they can think up all kinds of ridiculous things for you to script!)

- Easily modified—Depending on how you write your script, you can easily make a script capable of change (such as running on an alternative network). In addition, when a change is requested, a well-organized script can make the process painless.

☞ One way you can make your script easy to modify is to use environment variables and to set your own variables for anything that could change down the road. Use of comments and in-line documentation can make your script easy to follow. Additionally, when you segment your scripts by functionality, you can more easily determine where newly requested actions could best be inserted.

You could spend weeks or months perfecting a script to make it look nice and take every situation into account. This effort is well worthwhile at times and sometimes it is getting carried away. Establish a set of goals for your script and try your best to stick to accomplishing them. Additionally, documentation, support, and the potential to do a great deal of damage must be taken into consideration as potential drawbacks of scripting:

- Documentation—Administrators are not typically fond of documentation. However, an undocumented script can lead to problems when the individual that wrote the script is no longer around to answer questions. I have seen months of effort scrapped simply because it is easier to start over than to figure out what the existing script is doing (and why).

☞ Every scripting language available offers some form of support for comments or remarks. These are lines in the script that are not processed when executed and that you can use for in-line documentation. Regardless of whether you plan to start generating text or Help files to explain and instruct others about your scripts, in-line comments will help others to quickly understand what is happening and will also help you when you come back to a script you wrote long ago.

- Widespread damage—The potential to do damage is as much a problem for the experienced scripter that makes a simple mistake as it is for the newbie scripter that wants to try out a new script function. Particularly when dealing with scripts that reach out across the network to many machines, it is very important that operations be tested in a lab or offline environment before letting them loose on your production network.

Many environments restrict the use of development tools in production to prevent damaging mistakes but allow scripting to take place on the production network. This type of policy doesn't make sense—you can do just as much damage with a VBScript as you can with a Visual Basic application (see the sidebar “Script vs. Application: What is the Difference?”).

Script vs. Application: What is the Difference?

There is a fine line between script and application. Some organizations define applications as executable files with graphical user interfaces (GUIs) that are executed by users. A script can certainly meet these criteria. For example, a KiXtart script could be bundled into an executable with the KiXscripts Editor (<http://www.kixscripts.com>), utilize COM to provide a graphical interface (<http://www.kixforms.com>), and, with a simple shortcut, be made available for users to execute. Figure 7.1 shows a KiXtart script that uses KiXforms to provide users with a means of browsing Windows Management Instrumentation (WMI) and its values.

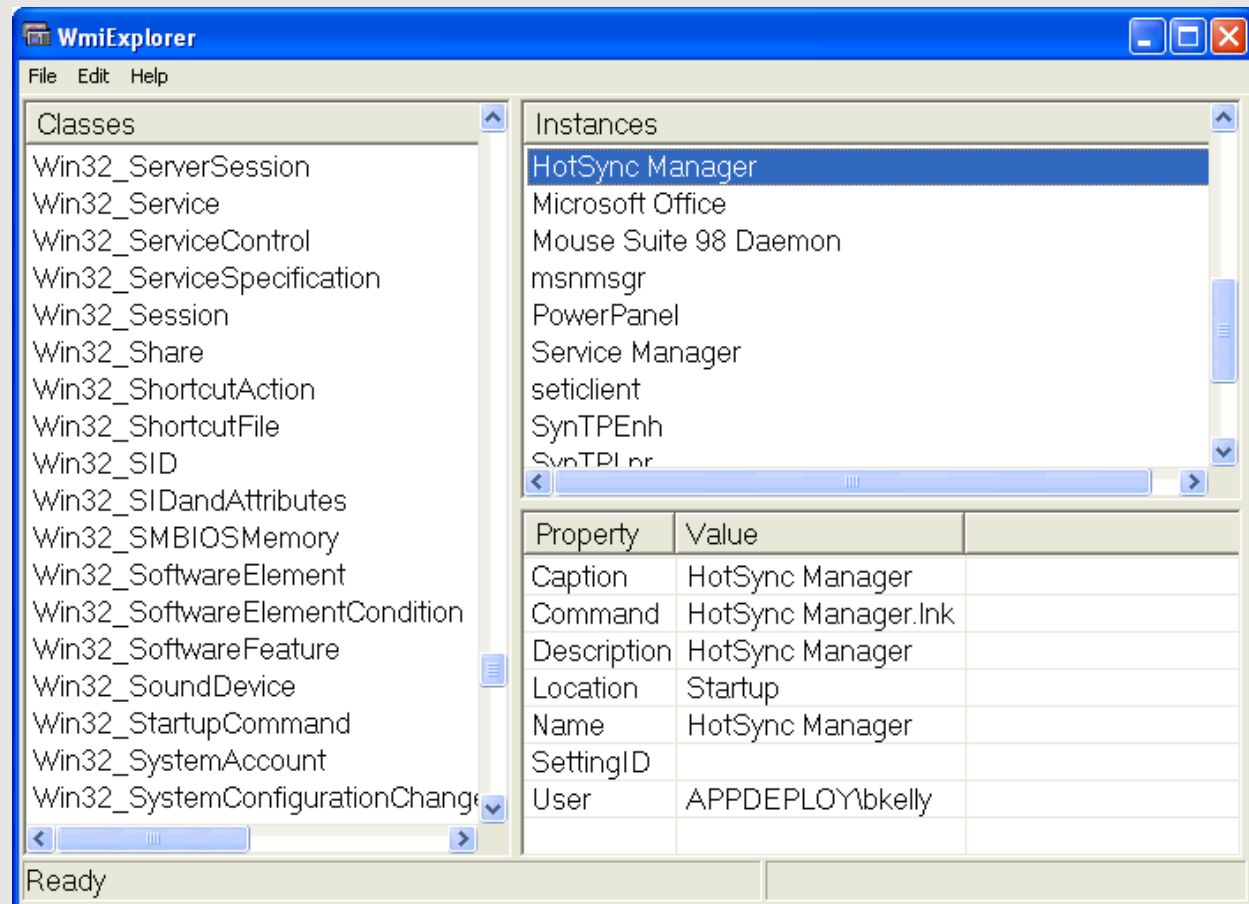


Figure 7.1: WMI Explorer script using KiXforms.

☞ An offline test or lab environment is ideal for script development. If you need access to a database or server-side component, consider replicating a minimal facsimile of what you require for your testing before using your script on a mission-critical server. If you don't have an entire lab at your disposal, consider products such as VirtualPC or VMware to emulate an entire network on your computer.

- Support—Using a third-party solution to perform a scriptable action ensures a certain level of support. If a change to the Windows OS or its security stops a custom script from functioning, who can you call?
- Upgrades and enhancements—Chances are, when you develop a custom script, it is with a very narrow view. You know exactly what you want the script to do, and you target the script to accomplish that goal. A third-party tool is typically enhanced and upgraded to remain competitive with similar tools. This competitive nature will likely result in the inclusion of features you would not have thought about nor had the time to develop.

Choosing a Scripting Language

There are many scripting languages available, and everyone has a preference. Like those that love Linux, Macintosh, or Windows, people generally have a strong opinion regarding the language they favor. Typically those administrators that learn a scripting language stick with it, and most administrators can prove that with enough effort, most any language can be made to accomplish much the same tasks. Although one language might be easier to learn or work with than another, completing a scripting task is really a matter of familiarity and creativity with whichever language you choose. For each of the languages presented in this chapter, I provide a sample for how to go about writing a value to the registry in an effort to help illustrate the complexity of each.

VBScript

VBScript is among the most popular scripting languages. It is still heavily used in Web sites that utilize Active Server Pages (ASP). With the introduction of the Windows Script Host (WSH), VBScript gained considerable exposure in the world of desktop administration. In addition, Microsoft routinely provides script examples using VBScript. VBScript makes heavy use of COM (which may be challenging to beginners), but there are several books and examples to help administrators become familiar with its use. The following example shows how to write a value to the registry using VBScript:

```
Set WSHShell = WScript.CreateObject("WScript.Shell")
WSHShell.RegWrite "HKLM\Software\BuildInfo\BuildNumber", "1.3A"
```

 You can download VBScript from <http://msdn.microsoft.com/downloads/list/webdev.asp>.

KiXtart

KiXtart was originally developed as a logon script processor. However, with each release, it offers more robust support, making it useful in most any situation. In addition to full COM support, KiXtart provides a number of built-in functions and variables that greatly simplify the scripting process when compared with performing those same actions using COM. Additionally, only the KiXtart executable is required (locally or on a network share) to execute KiXtart scripts in NT and later environments. KiXtart provides full support for Windows 9x as well (it requires a service to be run on one or more network servers in order to provide certain functionality). Its power, ease of use, and support for all versions of Windows make it very popular among administrators. The following example shows how to write a value to the registry using KiXtart:

```
$rc = WriteValue ("HKLM\Software\BuildInfo", "BuildNumber",
"1.3A", "REG_SZ")
```

 KiXtart is available for download from several KiXtart-dedicated sites, including <http://scriptlogic.com/kixtart>, <http://www.kixscripts.com>, and <http://www.kixtart.org>.

KiXtart Script Examples

As a result of its ease of use and robust capabilities, I have chosen KiXtart as the language to be used in script examples for this book. I did so for those not familiar with scripting, as KiXtart is an easy language to read and will help everyone more easily follow the examples provided. When it comes to administrative scripting, KiXtart is simple—especially when compared with most other languages. For those familiar with COM scripting, KiXtart provides a (COM) support—which will be familiar to those that know VBScript—with its CreateObject and GetObject functions.

KiXtart scripts are simple text files that you can write using any simple text or script editor. To run a KiXtart script, simply use your script file as a parameter when calling the KiXtart script processor (KIX32.EXE or wkIX32.exe) as the following line shows:

```
KIX32.EXE MyScript.kix
```

The KiXtart script processor may be called from a local or network location. Although the extension used is of no consequence, the .kix extension is typically used to identify KiXtart scripts. On Windows 9x clients, certain functionality is available only by installing the KiXtart RPC service on a network server. For more information, see the documentation included with KiXtart.

JScript

JScript (Java Script) provides similar capabilities to that of VBScript but with different formatting rules and built-in commands. JScript is Microsoft's implementation of JavaScript and is available as part of the Microsoft Windows Script. JScript relies heavily on COM. The following example shows how to write a value to the registry using Jscript:


```
sh = new ActiveXObject ("WScript.Shell");
sh.RegWrite ("HKLM\\Software\\BuildInfo\\BuildNumber", "1.4A");
```

 JScript is available as part of the Microsoft Windows Script at <http://msdn.microsoft.com/downloads/list/webdev.asp>.

WinBatch

Like KiXtart, WinBatch is a script language that gets a lot of attention for its ease of use. It is a powerful and fairly easy-to-learn language that has a dedicated following. WinBatch is not free, but a 30-day evaluation copy is available for download. Also available for WinBatch is a compiler that allows you to create freely distributable EXE files. The following example shows how to write a value to the registry using WinBatch:


```
RegSetValue (@REGMACHINE, "Software\BuildInfo\[BuildNumber]",
"1.3A")
```

 WinBatch is available for download at <http://www.winbatch.com>.

PerlScript

PerlScript (available within the free ActivePerl) is typically a language used by those already very familiar with Perl. It is a fairly complex scripting language and is not recommended for those that do not already have a strong background in Perl. Although not necessarily a benefit when it comes to administrative scripting, ActivePerl is provided for Linux and Solaris as well as Windows. Although the development tools provided are not free, the ActivePerl distribution is. The following example shows you how to write a value to the registry using PerlScript:

```
use Win32::Registry;
if ($HKEY_LOCAL_MACHINE->Create("SOFTWARE\\BuildInfo",$Key))
{
    $Key->SetValueEx("BuildNumber", 0, REG_SZ,"1.3A");
    $Key->Close();
}
```

 ActivePerl for Windows is available for download at <http://www.activestate.com/Products/ActivePerl>.

Shell Scripting

Most network administrators are familiar with shell scripting, commonly known as batch files. This widely used scripting language is often combined with other scripting languages. Although shell scripting has the most limited command set of the scripting languages, when used with other command-line tools (such as those from the Windows resource kits), shell scripting can be quite effective. To use shell scripting to write a value to the registry, you must call an external command-line utility.


COM Automation


Although the scripting language you choose will provide varying degrees of control over the functions you desire, most everything else can be handled via COM. COM provides the ability to share reusable code between vendors' applications as well as to expose certain functionality to other applications and scripts. Without getting too detailed about how COM works and because access to COM objects differs slightly from language to language, we will briefly cover a few COM objects that are commonly used in scripting.

 For more information about COM, visit http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/com_757w.asp.

WMI

WMI is Microsoft's implementation of Web-Based Enterprise Management (WBEM). WBEM establishes an architecture for supporting network enterprise management. WMI provides the ability to gather a great deal of information about a computer including installed software, peripherals, and other system details. WMI does not (at least very well) access information stored in AD.

 If you like to learn by example, check out Microsoft's clever Scriptomatic. This tool generates VBScript samples for a chosen class of WMI data. The Scriptomatic tool is available at <http://www.microsoft.com/technet/scriptcenter/WMI/matic.asp>. The PrimalScript script editor available at <http://www.sapien.com> provides a similar capability through its WMI script wizard feature.

 Check out the following sites for more information about WMI:

 WMI Tutorial at <http://www.microsoft.com/downloads/release.asp?releaseid=12570>

 WMI Scripting Primer at <http://msdn.microsoft.com/library/en-us/dnclinic/html/scripting06112002.asp>


 Microsoft WMI Scripting home at <http://msdn.microsoft.com/library/en-us/dnwmi/html/wmascript.asp>

Active Directory Service Interfaces

Active Directory Service Interfaces (ADSI) is a set of ActiveX controls (COM objects) that present the capabilities of directory services from different network providers as a single set of directory service interfaces. You can use ADSI to manage network resources in a directory service regardless of which network environment contains the resource.

WSH

Many people perceive WSH as a scripting language. WSH actually allows Windows systems to execute VBScript and JScript scripts. Additionally, WSH provides a set of components that expose functions helpful to scripting. You can take advantage of WSH using any language that supports COM automation.

 WSH is built-in to Microsoft Windows 98, Windows ME, and Win2K and later editions. For support for Windows 95 and NT 4.0 or to update to the latest version, visit <http://msdn.microsoft.com/scripting>.

ActiveX Data Objects

ActiveX Data Objects (ADO) is a collection of objects that provide an API to query data. Often used with Microsoft Access databases, ADO can also be used with different drivers and database engines to help you extract information from MS SQL, FoxPro, Oracle, and other sources, including Microsoft Word documents. The KiXtart script that Listing 7.1 shows uses ADO to record to an Access database the amount of free drive space on the local machine.

```

$CNstring = "provider=microsoft.jet.oledb.4.0;data source=" + $DBpath +
";persist security info=false"
$CMDtxt = "select * from COMPUTERS where computername = '" + @WKSTA +
""
$cn = CreateObject ("ADODB.Connection")
$cmd = CreateObject ("ADODB.Command")
$rs = CreateObject ("ADODB.RecordSet")
$cn.connectionstring = $CNstring
$cn.Open
$cmd.activeconnection = $cn
$rs.cursortype = 3 ; a snapshot of the recordset, not dynamic
$rs.locktype = 3 ; optimistic locking, assumes it is the only source
locking the record.
$rs.activecommand = $cmd
$cmd.commandtext = $CMDtxt $rs.Open ($cmd)

$DriveSpace = GetDiskSpace( "C:\" )

If $rs.eof = -1 $rs.addnew EndIf


$rs.fields.item("computername").value = @WKSTA
$rs.fields.item("domain").value = @DOMAIN
$rs.fields.item("drivespace").value = $DriveSpace
$rs.update

$rs.Close
$cn.Close

```

Listing 7.1: An example KiXtart script that uses ADO to record to an Access database the amount of free drive space on the local machine.

First, a connection string is established (\$CNstring) that identifies the type, source, and security settings for the connection to the database. The next variable, \$CMDtxt, is a SQL query string used to look for an entry in the database for the local workstation (@WKSTA is a KiXtart macro variable that stores this value). Next, we create the necessary objects and open the connection to the database. The connection is then set as the active connection so that we make use of it. The cursor type is set to three, which indicates a snapshot of the recordset (it is not dynamically updated), and the lock type is set to three, which indicates optimistic locking (assumes it is the only source trying to lock the record). The SQL query is then executed, and the KiXtart GetDiskSpace function is used to obtain the amount of free space for the C drive. If the SQL query fails to find a record that matches this computer, \$rs.eof (the end of file for the recordset) will be set to negative one. If such is the case, we use the addnew method to identify this record as a new database record before updating the values in the database. Each value is written to the database, and the update method is used to apply the changes to the database. Finally, the recordset and database connection is closed.

 One of the most common problems with the use of databases in scripts is the creation of a valid connection string. For details about the formatting of connection strings, see the Microsoft article "HOW TO: Create an ADO Connection from a Data Link File in Data Access Components" at <http://support.microsoft.com/default.aspx?scid=kb;en-us;300261>.

Internet Explorer

Internet Explorer (IE) exposes many functions to COM and many environments take advantage of the capabilities IE provides. The following two examples show how you can both automate actions and use IE to display information about the execution of your script.

Automate Actions

The following KiXtart script uses IE to navigate to the AppDeploy.com Web site:

```
$ie = CreateObject ("InternetExplorer.Application")
$ie.Visible = 1
$ie.navigate ("http://www.appdeploy.com")
```

First, the CreateObject function is used to create a session of the IE application in memory. As with many such objects, it is not visible to the user by default. If you were watching the Windows task list, you could observe a new instance of IE appearing as a process. This invisibility to the user is useful for automating tasks that users don't need to see; however, in this case, we want to see the action, thus the visible property for the IE object is set to true (a Boolean value where one is true and zero is false) so that the application is visible. Finally, the navigate method is used to instruct IE to navigate to the specified location—in this case AppDeploy.com.

Display Information

When most people think of a script, a DOS console window with text scrolling by comes to mind. Although there are certainly ways to make the information in the console window more attractive, you can utilize another method of displaying information that may be more "attractive" for your users. As the following script shows, IE can do the trick quite well (see Listing 7.2).

```

$ie = CreateObject ("InternetExplorer.Application")
$ie.AddressBar = 0
$ie.MenuBar = 0
$ie.StatusBar = 0
$ie.ToolBar = 0
$ie.Resizable = 0
$ie.Height = 450
$ie.Width = 400
$ie.Visible = 1
$ie.Navigate("about:blank")
While $ie.busy <> 0 AND @error = 0 Loop
$Doc = $ie.Document
$rc = $Doc.Open
$Doc.Write("<TITLE>Logon Window</TITLE>")
$Doc.Write("<BODY>")
$Doc.Write("Mapping Network Drives...<P>")
Sleep 2
$Doc.Write("Checking For Software Updates...<P>")
Sleep 2
$Doc.Write("Done!<P>")
Sleep 2
$rc = $Doc.Close
$ie.Quit
$ie = 0

```

Listing 7.2: Displaying information through IE.

In Listing 7.2, an object handle to the IE application is established so that we may begin addressing it. To keep IE from looking so much like IE, we hide the address bar, menu bar, status bar, and toolbar by setting them to false (zero). So that the display cannot be resized, we set resizable to zero as well, and establish the dimensions of the display with height and width values of 450 and 400, respectively. With this done, we make IE visible to the user by setting this property to one (true). We then navigate to a blank page, which is identified by the title about:blank. Because we do not want to start trying to write to IE before it has established the blank page for us, we loop until IE is no longer busy. We then open a new document and begin writing HTML to it using the write method. To keep the example short, we write a status message to IE, then set the script to sleep (pausing) a couple of seconds before displaying the next message. When done, we close the IE document and quit IE to destroy the IE object. Failure to do so will result in the window staying open.

Failing to close an object when complete can result in a new instance of the application being loaded into memory each time (as Figure 7.2 shows). Using a close or quit method is a common means of properly closing a document or application. In most cases, you can close objects by setting them to zero.

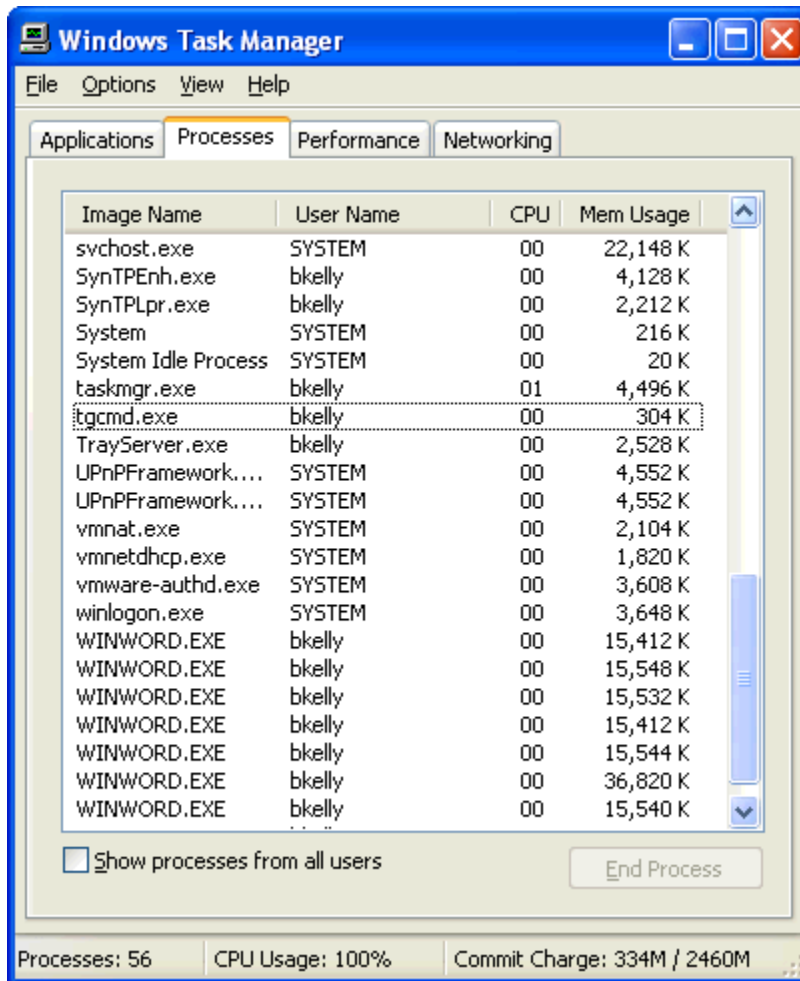


Figure 7.2: Task Manager shows multiple instances of Microsoft Word because it was not properly closed.


Running Your Scripts

Once you have created your script, how will you go about launching it? The answer is largely dependant upon the task you are trying to accomplish. If your script targets a user, a logon script, startup shortcut, or RunOnce registry entry is typically used. If you are targeting workstations, Group Policy, the scheduler service, and manually executed scripts are common choices. Let's explore each of these options.

Logon Scripts

Logon scripts are executed at logon before RunOnce registry entries and startup shortcuts. Unless third-party tools or utilities are being used, logon scripts always run in the security context of the user logging onto the system.

Logon scripts may be configured in the user profile or assigned to users by using Group Policy. Logon scripts are typically placed in the replicated NETLOGON share on a Windows network (%WINDIR%\system32\Repl\Export on NT and %SystemRoot%\Sysvol\Sysvol\\Scripts. on Win2K and later).

 If you don't specify a path, KiXtart searches files in the following order: NETLOGON, KiXtart Startup Directory, current directory.

For a Win2K client, the logon script runs minimized by default. To change this behavior, you must instruct the system to run logon scripts synchronously. Doing so will also cause Windows Explorer to not start until the logon script has finished running. Aside from making the logon script visible, this setting ensures that logon script processing is complete before the user starts working, but it can delay the appearance of the desktop.

If the Win2K client is a member of a Win2K domain, you can specify logon scripts should run synchronously by configuring the following setting in Group Policy:

```
User Configuration\Administrative Templates\System\Logon/Logoff
```

If the Win2K client is a member of an NT 4.0 domain, you can use System Policy to configure the following setting:

```
Computer Configuration\Administrative Templates\System\Logon
"Run logon scripts synchronously"
```

For Windows NT 4.0 or to make this change manually, set the value `RunLogonScriptSync` to a `REG_DWORD` value of 1 in the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon` hive. Alternatively, you can use a script to modify the `HideLogonScripts` registry value of type `REG_DWORD` and set the value to 0 in the `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\System` hive.

Logon scripts are most often used to perform dynamic “per user” actions when a user logs onto the network. For example, Listing 7.3 shows a logon script that sets drive and printer mappings.

```
If InGroup("HR")
    Use P: "\\fileserver\hr_files"
EndIf
If InGroup("Managers")
    Use M: "\\fileserver\managers"
EndIf

$Floor = Left(@WKSTA,2)
Select
    Case $Floor = "01"
        $rc = AddPrinterConnection("\\printserver\floor1_prnt")
    Case $Floor = "02"
        $rc = AddPrinterConnection("\\printserver\floor2_prnt")
    Case $Floor = "03"
        $rc = AddPrinterConnection("\\printserver\floor3_prnt")
EndSelect
```

Listing 7.3: An example logon script that sets drive and printer mappings.

This script maps the P drive and M drive for those in the HR and Managers user groups, respectively. If a user is not in either group, neither of these drives will be mapped for them. If a user is in both groups, both drives will be mapped by this script. The script assumes that the two leftmost characters in the computer name identify the floor where the computer resides. A different printer is mapped depending upon which floor the computer is located.


RunOnce and Run

The RunOnce and Run keys may exist for both the machine (HKEY_LOCAL_MACHINE) and the current user (HKEY_CURRENT_USER). Any machine RunOnce values execute prior to the loading of the desktop. Then after the desktop loads, any values specified in the Run keys for the machine and the user are then executed (respectively), and finally RunOnce for the current user is executed.

The specified commands will be executed in the context of the user logging on to the system. Although Run values remain on the system and execute during each logon, RunOnce values execute and then remove themselves from the RunOnce key so that they do not run again:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
```


To specify an entry, simply create a value in the appropriate key, and give the value any name you like (the value name does not matter, but should be something to help you identify its action). Then, as the data for the new value, enter the command line or script path that you want to execute.

 Never call a script from RunOnce that inserts an entry into RunOnce or you will encounter an endless loop. RunOnce does not continue the logon process until all entries are executed and the RunOnce key in the registry is free of entries.

Startup Shortcuts

Startup shortcuts are executed after the logon script and any calls made by Run and RunOnce values. If there are multiple items in the startup folders for the current user, those items are triggered alphabetically starting with those assigned to all users. The following list shows the order of startup execution:

1. Network logon script
2. HKEY_LOCAL_MACHINE RunOnce entries
3. HKEY_LOCAL_MACHINE Run entries
4. HKEY_CURRENT_USER Run entries
5. HKEY_CURRENT_USER RunOnce entries
6. All users' startup folder shortcuts
7. Current user startup folder shortcuts

 For a useful article about startup sequences, including additional actions and entry points into the logon process, see the ScriptLogic article T1056 at <http://www.scriptlogic.com/support/kb/displayarticle.asp?UID=66>.


Schedulers

You can use the following schedulers to configure events to occur at specified times:

- **At.exe**—At.exe is a command-line scheduler that Microsoft introduced with NT. This scheduler provides the ability to schedule events to occur on either a local or remote system—a very valuable desktop administration tool. For example, the command

```
AT 20:00/every:m,t,th,f "C:\Chk4Updt.bat"
```

causes C:\Chk4Updt.bat to be run each Monday, Tuesday, Thursday, and Friday at 8:00 PM.

 When Microsoft released IE 4.0, at.exe produced inconsistent or unexpected results. The problem was that the Task Scheduler replaced the Scheduler service and did not provide the same support at the command line. IE 5.01 SP1 corrected the problem.

- **Soon.exe**—Soon.exe is referred to as a near-future command scheduler. It utilizes the at.exe scheduler to set the specified event to occur a couple of seconds in the future. With at.exe, it is tricky to schedule an event to occur right away—if you are off by just a few seconds, the task is set to run the next day. On top of this problem, you cannot assume that your system clock and the remote system clock have the same time configured. Soon.exe takes care of this problem. For Windows XP and later, the SchTasks.exe utility supports a parameter to identify a task that should be run immediately.

 Soon.exe is available in NT and later resource kits as well as online at <http://www.microsoft.com/windows2000/techinfo/reskit/tools/existing/soon-o.asp>.

- **SchTasks.exe**—Windows XP and Windows Server 2003 include SchTasks.exe, which replaces the NT and Win2K command-line scheduler at.exe. SchTasks.exe allows an administrator to create, delete, query, change, run, and end scheduled tasks on a local or remote system. The following command-line parameters control the SchTasks.exe command-line utility:

```
/Create to create a new scheduled task
```

```
/Delete to delete a scheduled task
```

```
/Query to display scheduled tasks
```


```
/Change to change the properties of a scheduled task
```

```
/Run to run a scheduled task immediately
```

```
/End to stop the currently running scheduled task
```


- **WMI**—You can use WMI to manipulate the Windows Task Scheduler, though the format for identifying dates and times is rather complex. The time to run the job must be specified in the form of YYYYMMDDHHMMSS.MMMMMM(+-)OOO. The date (YYYYMMDD) must be replaced by ********* because the scheduling service only allows jobs to be configured to run one time or run on a day of the month or week (and cannot specify a specific date). In the following KiXtart sample code, a utility named DiskCheck is configured to run every Wednesday at 4:25 AM:

```
$Computer = "."
$WMIService = GetObject("winmgmts:" +
"{impersonationLevel=impersonate}!\\" + $Computer +
"\root\cimv2")
$NewJob = $WMIService.Get("Win32_ScheduledJob")
$src = $NewJob.Create ("Notepad.exe", "*****012500.000000-420",
True , 4, , True, JobID)
```

 For more information about using COM to schedule jobs (including how to decipher these time and date values) visit http://msdn.microsoft.com/library/en-us/wmisdk/wmi/win32_scheduledjob.asp.

Manual Execution

To configure your script to perform actions on remote systems, simply running the script from your workstation may be all you need to do. Any security restrictions for remote connections and rights will need to be taken into consideration. In most cases, executing an administrative script while logged on as a member of the Domain Admins group is more than sufficient to perform most all actions you want to automate. However, it is always best to utilize an account that has just enough permissions to get the job done.


 Always be aware of the potential for scripts to result in widespread damage. When it comes to scripting, it is very important that operations be tested in a lab or offline environment before letting them loose on your production network. Such is especially the case when dealing with scripts that reach out across the network to many machines.

To access a file remotely on NT and later systems, you can take advantage of administrative shares. For example, to copy a file locally, you might use something like this:

```
Copy "\\server\share\file.exe" "c:\winnt\system32"
```

To copy the file to a remote machine, you can take advantage of administrative shares by changing this statement to the following:

```
Copy "\\server\share\file.exe" "\\computername\admin$\system32"
```

 Depending upon System Policy or Group Policy settings in your environment, these administrative shares (\$Admin, \$C, \$D, etc.) may be disabled.

In KiXtart, you can access remote registries in much the same way by preceding with the universal naming convention (UNC) path to the target system when identifying the subkey you want to work with (the computer name preceded by two backslashes). Of the following examples, the first example shows the reading of a local registry value, and the second illustrates the same action against a remote system by using its UNC name.

```
$EditorVersion = ReadValue
("HKLM\SOFTWARE\iTripoli\KixscriptsEditor", "Version")

$EditorVersion = ReadValue
("\\computername\HKLM\SOFTWARE\iTripoli\KixscriptsEditor",
"Version")
```

Group Policy

Group Policy can target machines or users at startup (before the logon prompt is presented) or during logon. Additionally, you can target scripts to specific domains, organizational units (OUs), or groups from within AD:

- Logon—A logon script runs in the user context as a user logs on to a workstation.
- Logoff—A logoff script runs in the user context as a user logs off of a workstation.
- Startup—A startup script runs in the context of the local system account when a computer is started, before the opportunity to log on is presented.
- Shutdown—A shutdown script runs in the context of the local system account when a computer is shutdown.

Assigning Scripts

On networks that have implemented AD, the preferred method of configuring logon scripts is to assign them via Group Policy. To do so:

1. Open the Group Policy snap-in (in Administrative Tools, Active Directory Users and Computers).
2. Right-click the OU (in the console tree) to which you want to assign the logon script. For all members of the domain, right-click the domain.
3. Select Properties, and select the Group Policy tab.
4. Create a new Group Policy Object (GPO) by clicking New or choose an existing GPO, and click Edit.
5. Double-click Logon in the right pane under User Configuration/Windows Settings/Scripts.
6. Click Add. Notice that a browse dialog box appears for selection of a file from within the GPO directory.
7. Copy and paste your script into this browse dialog box to make it available to the GPO, if you have not already done so. If you are using a scripting language that is not installed and requires a support file (such as KiXtart's kix32.exe), include this as well.

8. Select the script to be run as your script name. If you're using a language that uses an executable file to process the script, set that as the script name and use the parameters text box to enter the name of the script file. If you assign multiple scripts, the scripts are processed according to the order that you specify. To move a script in the list, click the script, then click either Up or Down.

Avoiding Repeated Executions

Because scripts assigned to a user will always execute, it is important to add a condition for execution when dealing with scripts that you do not want to run again and again. One method is to look for the result of the script you are trying to run. For example, if a logon script is to map a drive to a network share, the script can base this action on whether the drive is already mapped to the desired share. Another method is to drop a *footprint* that indicates an action has taken place. A footprint is a term used to indicate something left behind in this way (such as a file or registry entry).

Using a registry entry, the following KiXtart script checks to see whether the current date has been updated by this script today. If it has not, the script records the current date, then executes the script. When the script is executed again on the same day, the date will match and the script will not be called.

```
If ReadValue("HKCU\Software\Scripts","TimeCardReminder.kix") <> @DATE
    $src =
WriteValue("HKCU\Software\Scripts","TimeCardReminder.kix","@DATE","REG
_SZ")
    Call "TimeCardReminder.kix"
EndIf
```

Using a registry key, the script will check to see whether the script has been executed before. If the word Executed is not found in a registry value being used to identify a script that should only be run one time, the script is executed and the registry is updated to reflect it. When run in the future, the value will exist, and the script call will always be skipped:

```
If ReadValue("HKCU\Software\Scripts","Acknowledgement.kix") <>
"Executed"
    $src =
WriteValue("HKCU\Software\Scripts","Acknowledgement.kix","Executed","R
EG_SZ")
    Call "Acknowledgement.kix"
EndIf
```

Building Lists

Tackling a project is the best way to get started in scripting, so let's get right into it with a common first step in many administrative scripts, building a list. To run an operation against several client workstations, it is common to base such actions against a list of machines. Although the same script can create the list and perform the desired action, it is common to dynamically perform actions against the list gathered without actually writing the machines to a list file. However, to keep the examples in this book shorter and more useful, I will separate the task of creating a list of machines with that of the scripts that follow, which will utilize the lists.

 For a comprehensive look at KiXtart, see *The Start to Finish Guide to Scripting with KiXtart* at <http://www.kixscripts.com/book>.

Machine List

Depending upon your environment, you can go about creating a machine list in a number of ways. In an AD environment, you would probably utilize your OU structure to target systems. You may also make use of the Windows network browse list to retrieve workstation names. I'll discuss and show you examples of these options in the following sections.

AD

For organizations that have taken on the effort of creating an AD domain, there is typically extensive planning involved. Therefore, making use of that planned structure is often a worthwhile approach to targeting systems with scripts. Aside from utilizing Group Policy to assign scripts to desired systems, you can use your AD structure to build a desired list of machines, which you can then use to perform actions on computers remotely from a manually executed script. For example, in the script that Listing 7.4 shows, KiXtart is used to write the machines found in AD to a file named `machines.txt` in the Windows Temp folder.

```
$rc = Open (1, "%TEMP%\Machines.txt", 5)
  $Container = GetObject ("LDAP://cn=computers, dc=intranet,
dc=appdeploy, dc=com")
  For Each $Computer In $Container
    If $Computer.class = "Computer"
      $ComputerName =
SubStr($Computer.Name, 4, Len($Computer.Name))
      $ = WriteLine (1, $ComputerName + @CRLF)
    EndIf
  Next
$rc = Close (1)
```

Listing 7.4: Using KiXtart to write machines found in AD to the `machines.txt` file in the Windows Temp folder.

In the open statement, the number one is a handle used to identify this file later in the script. The number five tells KiXtart to create the file if it does not already exist and to open the file for writing. An object is created to the computer container, as specified in the LDAP string (for information about how to create an LDAP query string, see the sidebar “Constructing an LDAP Query String”). The class property identifies whether a computer is a computer, group, and so on. Because the computer name is returned as `CN=MyComputer`, we use the `SubStr` function to create a new string (`$ComputerName`) to reflect all characters from the fourth character to the end of the string (thereby stripping the `CN=` portion of the name from the string). Next, we write this to the file opened earlier as 1, and include the macro variable `@CRLF`, which identifies a carriage return and line feed (otherwise all computer names would appear on one long line). Once each item in the `$Container` object is processed, the `machines.txt` file is closed.

Constructing an LDAP Query String

For beginners, the most difficult part about this script is getting the LDAP query string format correct. This task is not so difficult once you do it a few times.

```
LDAP://
```

specifies the provider type.

```
cn=computers
```

refers to the default container in AD. I chose this container because it is where all computer objects are placed by default. If you want to use actual OUs, then the `cn=` must change to `ou=`. The `cn=` is only for the default containers. You can distinguish the default containers, as they don't have the OU icon on them (unlike the ones you create). If the container you want to specify is more than one level deep, you must always start with the most specific container first. For example, `ou=subOU, ou=higherlevelOU`.

```
dc=
```

specifies the DNS domain name. As with OUs, each level must start with `dc=`, and each level must be comma separated. For example, `intranet.appdeploy.com` would be entered as

```
dc=intranet,dc=appdeploy,dc=com
```

Server Manager

For NT domains (as well as Win2K and Windows Server 2003 servers), you can query servers for computer names using a method similar to that of AD by using the WinNT provider. Although it does not provide many of the details about objects that LDAP does, this method is easier to use. Its syntax does not require you to deal with the hierarchical structure of AD, and it can be used to access domain-based SAM databases (residing on NT 4.0 domain controllers), local-based SAM databases (for NT 4.0 and Win2K member servers and workstations), and Win2K domain controllers. In the example that Listing 7.5 shows, KiXtart is used to create a list of the machines found in the current domain's SAM database (in KiXtart, `@DOMAIN` is used to represent the current domain name).

```
$rc = Open (1, "%TEMP%\Machines.txt", 5)
$Domain = GetObject ('WinNT://' + @DOMAIN + ',domain')

For Each $Computer In $Domain
    If $Computer.class = "Computer"
        $rc = WriteLine (1, $Computer.name + @CRLF)
    EndIf
Next
$rc = Close (1)
```

Listing 7.5: Using KiXtart to create a list of machines found in the current domain's SAM database.

In the open statement, the number one is a handle used to identify this file later in the script. The number five tells KiXtart to create the file if it does not already exist and to open the file for writing. If the class property identifies the object as a computer, it is written to the file opened earlier as 1, and includes the macro variable `@CRLF`, which identifies a carriage return and line feed. Once each item in the `$Domain` object is processed, the `machines.txt` file is closed.

NET VIEW

Using the NET VIEW command, you can produce a list of computers from the network browse list. The network browse list provides a list of machines, so the list is easily gathered; however, this method is not always the most reliable because (by default) the network browse list only contains systems that are (or have recently been) active on the network.

The script that Listing 7.6 shows uses the greater than symbol (>) to direct the output of the NET VIEW command to a temporary file that is used by the script to create a clean list of computers. Because the output of the NET VIEW command contains a header and computer comments, the script uses the InStr and SubStr functions from the third character until the first space is encountered on lines that start with \\. The computer names are written to the file opened as 2, and when complete, file 2 is closed. Finally, the temporary file containing the NET VIEW output is also closed, then deleted.

```
Shell "%comspec% /c net view /domain:" + @DOMAIN + "
>%temp%\netview.tmp"
$rc = Open (1, "%temp%\netview.tmp")
$rc = Open (2, "%temp%\machines.txt", 5)
$line = ReadLine (1)
While @error = 0
    If InStr ($line, "\\")
        $rc = WriteLine (2, RTrim (SubStr ($line, 3, InStr ($line,
" ") - 1)) + @CRLF)
    EndIf
    $line = ReadLine (1)
Loop
$rc = Close (2)
$rc = Close (1)
Del "%temp%\netview.tmp"
```

Listing 7.6: Using NET VIEW to create a machine list.

Custom Script Examples

You can script most anything; the following sections provide examples of scripts that are used to accomplish common administrative tasks. In the following sections, we will make use of the computer name lists generated to restart machines, backup their event logs, and gather software inventory information.

Restarting Remote Systems

Restarting machines is an action that is becoming increasingly popular to script as a result of Group Policy. With Group Policy, when new software is assigned to a machine, a reboot is necessary in order to trigger the installation process. In the example that Listing 7.7 shows, the script reads the machines.txt file, and for each machine included, issues a shutdown command. The syntax for the KiXtart shutdown command is to include the name of the machine to be shut down, the message to be displayed, and the number of seconds to count down before shutdown begins. The last two values specify whether the user should be forced off and whether the system should reboot. In the fourth parameter, a zero indicates that if the user has unsaved work, the user should be prompted to save before the script logs off the user (if it were set to one, the user would be forcibly logged off without prompting). The fifth and final parameter indicates the system should be restarted (if it were set to zero, the system would simply be shut down). The ReadLine function reads a line from the specified script with each use. The script loops until this function tries to read a line and fails (when the end of the list is encountered).

```
$src = Open (1, "%temp%\machines.txt")
$Machine = ReadLine (1)
While @ERROR = 0
$src = Shutdown ($Machine, "Restarting To Install New Software", 30, 0,
1)
    $Machine = ReadLine (1)
Loop
$src = Close (1)
```

Listing 7.7: Using a KiXtart script to restart remote systems.

Backing Up Event Logs

Managing event logs is yet another task best left to a script or other automated process. The script that Listing 7.8 shows reads each machine name from the machines.txt file and uses the BackupEventLog function to create a backup of the security log on a server share. If the task was successful, it will return an error level of zero (in KiXtart, you learn the returned error by looking at the @Error macro variable). If the variable is zero, we use the ClearEventLog function to clear the remote system's security log and move on to the next machine. When the end of the machine list is reached, attempting to read the next computer name will result in an error that will exit the loop, close the machines.txt file, and exit.

```
$src = Open (1, "%temp%\machines.txt")
    $Machine = ReadLine (1)
    While @ERROR = 0
        $src = BackupEventLog ("\\" + $Machine +
"\Security", "\\server\share\Application_Log_" + @WKSTA)
        If @ERROR = 0
            $src = ClearEventLog ("\\" + $Machine + "\Security")
        EndIf
        ? $Machine + " - " + @SERROR
        $Machine = ReadLine (1)
    Loop
$src = Close (1)
```

Listing 7.8: A script that backs up event logs.

Software Inventory

Software inventory is another task that can be performed a number of ways. We'll explore two methods here—reading the Uninstall key from the registry and querying the Windows Installer object with WMI.

Reading the Registry

The data presented in the Add/Remove Programs applet provides information about most every program installed on a system. Applications that do not provide an uninstall, such as simple command-line utilities, will not be detected; however, most applications that provide an installation program register themselves in the Uninstall subkey in the registry. Usually, each application provides its name and the command that is executed for removal (see Figure 7.3).

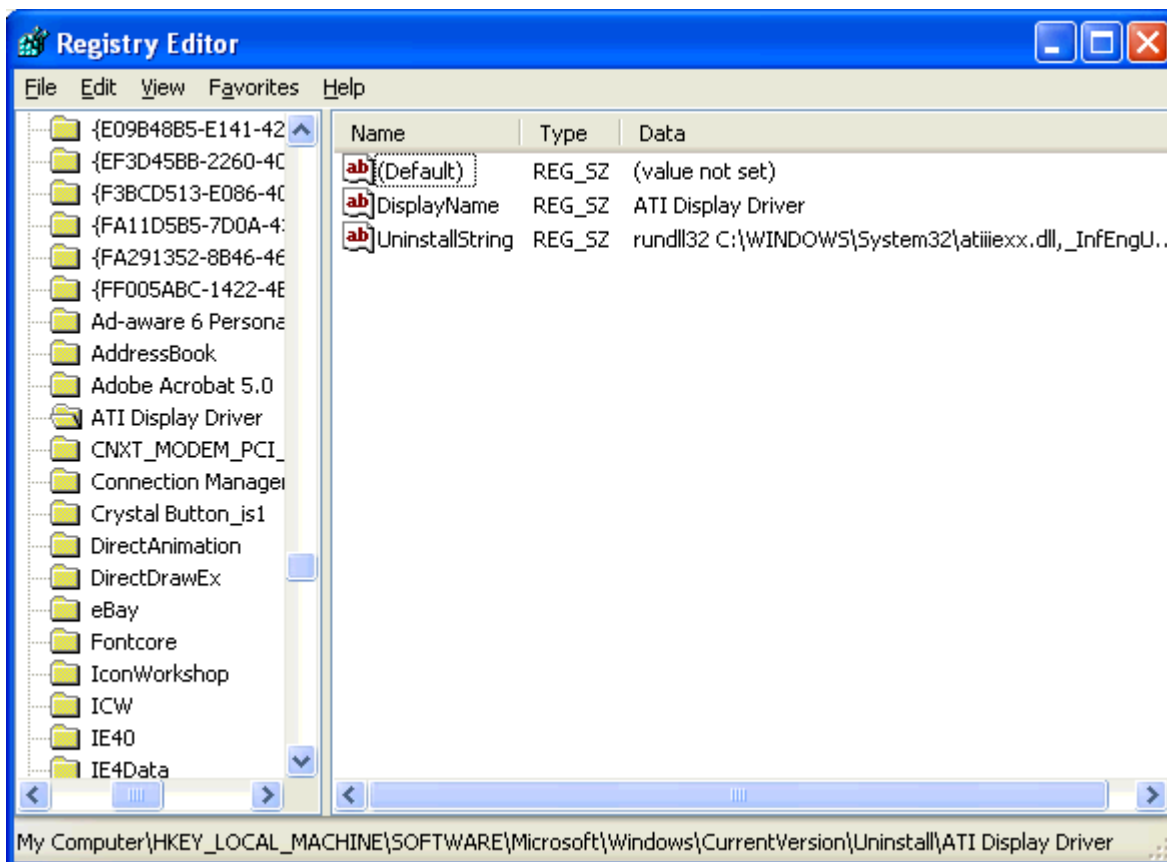


Figure 7.3: Uninstall subkey values contain a list of software installed on a system.

Although MSI setups provide a fixed set of values in the Uninstall subkey, those of non-MSI setups can be very inconsistent. In the example that Listing 7.9 shows, the script enumerates each of the values in the remote system's Uninstall key. Because we cannot count on a consistent value to identify the application, the script looks for DisplayName and then QuietDisplayName to identify the application. If neither are found, the name of its subkey within Uninstall is used.

```

$rc = Open (1, "%Temp%\Machines.txt")
$rc = Open (2, "%Temp%\Software.txt", 5)
$Machine = ReadLine (1)
While @ERROR = 0
    $rc = WriteLine (2, $machine + @CRLF)
    $UninstallSubKey = "\\\" + $MACHINE +
"\HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall"
    $Count = 0
    $Key = EnumKey ($UninstallSubKey, $Count)
    While @Error = 0
        $App = ReadValue ($UninstallSubKey + "\$KEY",
"DisplayName")
        If $App = ""
            $App = ReadValue ($UninstallSubKey + "\$KEY",
"QuietDisplayName")
        EndIf
        If $App = ""
            $App = $Key
        EndIf
        $rc = WriteLine (2, $App + @CRLF)
        $Count = $Count + 1 $Key = EnumKey ($UninstallSubKey,
$Count)
    Loop
    $rc = WriteLine (2, @CRLF)
    $Machine = ReadLine (1)
Loop
$rc = Close (2)
$rc = Close (1)

```

Listing 7.9: Reading the registry to determine the system's software inventory.

Windows Installer

WMI lets you query installed software. However, this method will only return applications installed with Windows Installer (MSI). In the example that Listing 7.10 shows, the script reads each machine name from machines.txt, and connects to the WMI object in order to query for installed software. Each application found is written to a text file.

```

$rc = Open (1, "%temp%\machines.txt")
$rc = Open (2, "%temp%\software.txt",5)
$Machine = ReadLine (1)
While @ERROR = 0
    $rc = WriteLine (2, $machine + @CRLF)
    $WMI = GetObject ("winmgmts:{impersonationLevel=impersonate}!\\\"
+ $Machine + "\root\cimv2")
    $Software = $WMI.ExecQuery("Select * from Win32_Product")
    For Each $Application In $Software
        $rc = WriteLine (2, $Application.Name + @CRLF)
    Next
    $rc = WriteLine (2, @CRLF)
    $Machine = ReadLine (1)
Loop
$rc = Close (2)
$rc = Close (1)

```

Listing 7.10: Using WMI to inventory a system's software.

The AutoLogon Process

AutoLogon provides you with the ability to have a machine automatically log on as a user so that you can interact with the desktop in a traditional fashion with your script. Some installations require that an administrator log on following the initial portion of the setup so that actions can take place following a reboot. It is also common to follow the imaging or unattended Windows installation process with a script that will perform additional installations and configuration changes.

There are several tools available to utilize an alternative set of credentials so that users triggering a process (such as through a logon script) can get past normal security restrictions. This method of utilizing the AutoLogon process is best suited for cases in which you cannot repackage or utilize silent switches to automate a task. In these situations, you might find it necessary to address prompts and dialog boxes with your script, which will require that a desktop profile be loaded. You can do so using the Windows resource kit ScriptIt tool or KiXtart's built-in SendKeys function. Both options require that dialog boxes be addressed by name, which requires that the dialog boxes exist on the desktop so that they may be addressed.

NT and later provide this capability by setting the following keys in the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon` subkey:

- AutoAdminLogon
- DefaultPassword
- DefaultUsername
- DefaultDomainName
- AutoLogonCount (Win2K and later)

Set AutoAdminLogon to 1 to instruct the system to attempt an automatic logon. The DefaultPassword, DefaultUsername, and DefaultDomainName values should be updated to include the account information with which you want to automatically log on. The downside is that the values are all in plain text and can be read by anyone who knows where to look. Thus, when you're completing your desired AutoLogon sessions, it is important to not only set the AutoAdminLogon value to zero but also delete the logon credentials stored in the other values. Win2K and later provide a useful feature to handle this cleanup for you—simply set the AutoLogonCount value to the number of AutoLogon sessions you want to initiate, and this value will be decremented with each automatic logon performed. When it reaches zero, it cleans up the logon credentials for you, and resets the AutoAdminLogon value back to zero.

In the example that Listing 7.11 shows, the system is configured to logon automatically two times with the AutoLogonCount value. The computer will attempt to log on to the appdeploy domain with a username of bkelly and a password of 12345678. Then the machine is restarted to begin the AutoLogon process.

```
$WinLogonSubKey = "HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon"
$rc = WriteValue ($WinLogonSubKey, "AutoLogon", "1", "REG_SZ")
$rc = WriteValue ($WinLogonSubKey, "DefaultUserName", "bkelly", "REG_SZ")
$rc = WriteValue
($WinLogonSubKey, "DefaultPassword", "12345678", "REG_SZ")
$rc = WriteValue
($WinLogonSubKey, "DefaultDomainName", "appdeploy", "REG_SZ")
$rc = WriteValue ($WinLogonSubKey, "AutoLogonCount", "2", "REG_SZ")
$rc = Shutdown("", "Shutting Down For AutoLogon", 10, 1, 1)
```

Listing 7.11: Scripting the AutoLogon process.

Restricting Access

Odds are that if you are having a script perform an automatic logon to the system, it is because you need to have the computer logged on as an administrator. To say the least, this is not likely something your security department will look kindly on without some measure taken to limit a user from interrupting the script and accessing the system with the administrative account. Locking the mouse and keyboard is a typical approach to addressing this issue. By disabling the mouse and keyboard devices, you can perform an automatic logon without worry of users interrupting the script.

Although you cannot stop the mouse or keyboard devices during a current Windows session, you can disable the devices so that they do not start the next time the system reboots. The device settings are located in the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services subkey. The mouse device value is MouClass and the keyboard device value is KbdClass.

As the AutoLogon process requires a reboot already, disabling these two devices before the restart fits nicely into the process. However, in order to restore the system, you will need to undo the changes when you are done. The script that Listing 7.12 shows configures a computer for automatic logon and disables the mouse and keyboard drivers for next logon (by setting their start values to 4). Before restarting the computer, a RunOnce entry is added to the system to perform actions and unlock the system in order to control the AutoLogon process.

```

$WinLogonSubKey = "HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon"
$rc = WriteValue ($WinLogonSubKey, "AutoLogon", "1", "REG_SZ")
$rc = WriteValue ($WinLogonSubKey, "DefaultUserName", "bkelly", "REG_SZ")
$rc = WriteValue
($WinLogonSubKey, "DefaultPassword", "12345678", "REG_SZ")
$rc = WriteValue
($WinLogonSubKey, "DefaultDomainName", "appdeploy", "REG_SZ")
$ServicesSubKey = "HKLM\SYSTEM\CurrentControlSet\Services"
$rc = WriteValue($ServicesKey + "\MouClass", "Start", "4", "REG_SZ")
$rc = WriteValue($ServicesKey + "\KbdClass", "Start", "4", "REG_SZ")
$RunOnceSubKey =
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce"
$rc =
WriteValue($RunOnceSubKey, "AutoScript.bat", "c:\temp\AutoScript.bat", "RE
G_SZ")
$rc = Shutdown("", "Shutting Down to Run AutoScript.bat as
Admin", 10, 1, 1)

```

Listing 7.12: Locking the mouse and keyboard and restarting for AutoLogon.

Listing 7.13 shows an example script that lets you restore the system. You would precede this code with whichever actions you want to automate as administrator. The batch file that was added to RunOnce in the previous example would call the following script to restore control of the system.

```

$WinLogonSubKey = "HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon"
$rc = WriteValue ($WinLogonSubKey, "AutoLogon", "0", "REG_SZ")
$rc = WriteValue ($WinLogonSubKey, "DefaultUserName", "")
$rc = WriteValue ($WinLogonSubKey, "DefaultDomainName", "", "REG_SZ")
$rc = DelValue ($WinLogonSubKey, "DefaultPassword")
$ServicesSubKey = "HKLM\SYSTEM\CurrentControlSet\Services"
$rc = WriteValue($ServicesKey + "\MouClass", "Start", "1", "REG_SZ")
$rc = WriteValue($ServicesKey + "\KbdClass", "Start", "1", "REG_SZ")
$rc = Shutdown("", "Shutting Down to Restore System Control", 10, 1, 1)

```

Listing 7.13: Unlocking the mouse and keyboard and restarting to restore control.

Summary

In this chapter, we covered some of the benefits and drawbacks to scripting as well as some tips about how to choose a scripting language. We discussed COM automation, and the ways you can use it to interact with directory services, databases, and applications. Some common ways to run your scripts were covered along with the order and security context for each. Finally, we went over some real-world implementations, including how to generate a list of machines on a network and ways to then use these lists to reboot computers, manage event logs, and gather software inventory. In the next and final chapter, we will discuss asset management—both software and hardware, including the benefits and tools available to do so.