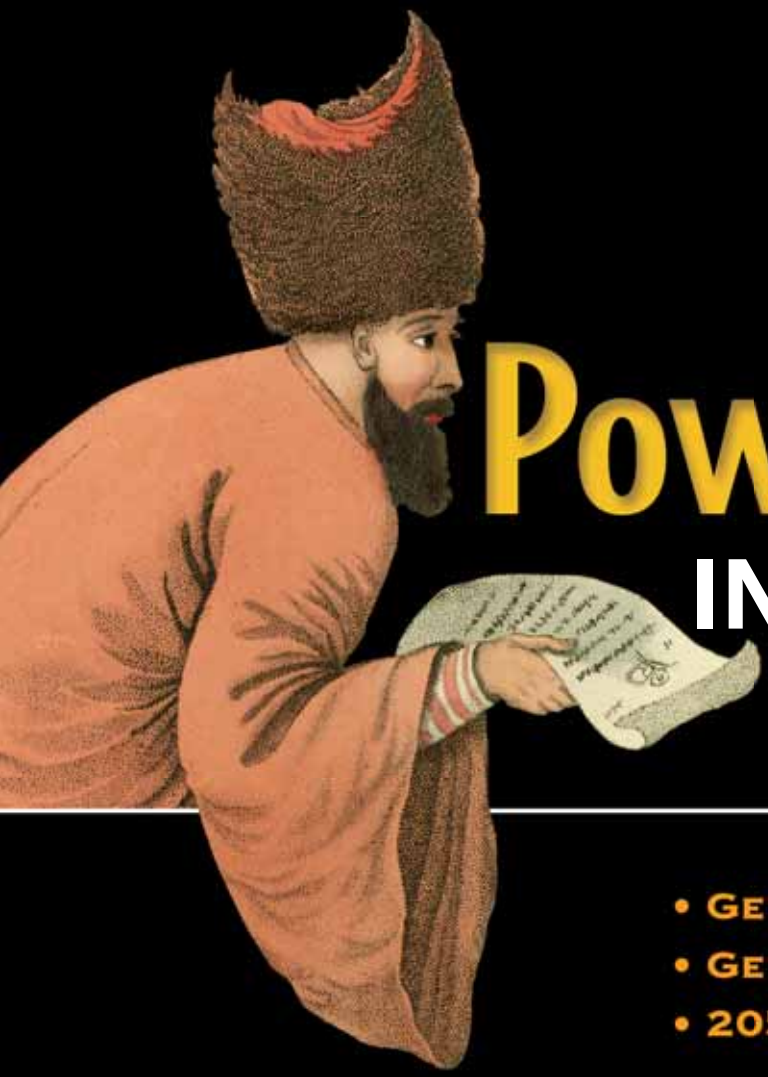
Richard Siddaway



PowerShell IN PRACTICE

- GET GOING
- GET SAVVY
- 205 PRACTICAL TECHNIQUES

SAMPLE CHAPTER

MANNING

PowerShell in Action

by Richard Siddaway

Chapter 7

brief contents

7 Desktop

This chapter covers

- Discovering a machine's configuration
- Testing printers and printer drivers
- Working with special folders
- Working with Microsoft Office applications such as Word and Excel

This chapter closes out the part of the book dealing with users and leads into chapter 8, which opens the server administration section.

> **HOW TO USE THE SCRIPTS** Many of the scripts presented in this chapter could be run interactively from the PowerShell prompt rather than as a script. If anything is run frequently, consider creating a function with the computer name as a parameter. Alternatively, PowerGUI could be utilized (see section 4.5.2).

Group Policy is used to configure and manage desktops in many corporate environments. Group Policy is a great technology that's underutilized in many cases. Group Policy objects (GPOs) are great for configuring the computer, but they don't

report back the actual configuration. There will still be a need to investigate desktop (or server) issues even with an extensive utilization of GPO-based technologies.

This chapter shows how to investigate those issues by discovering information about the computer, and how to configure aspects of the computer that can't be reached by other means. In this chapter, we'll make extensive use of the WMI and COM capabilities of the Windows environment. Using these technologies in PowerShell was covered in chapter 3. If you jumped straight into the sections of the book covering the scripts, it might be worth looking at chapter 3 as a refresher before delving too far into this chapter.

7.1 Automating desktop configuration

In most organizations, there are a lot more desktops (that does include laptops for this discussion) than servers. This means that there will be a lot more administrative effort spent on the desktop estate. Anything that can reduce that overhead will have a beneficial impact on the company. This is where automation comes into the picture. If I can make changes remotely or remotely discover information that will help me resolve the user's problems, I can be more productive. How can I be more productive? The answer in the Windows environment of today is to use PowerShell.

When we want to investigate an issue with a user's computer (or a server), we need to know how it's configured. The first part of this chapter uses PowerShell and WMI to discover configuration information. A number of scripts are presented that show how to go about discovering this information. Rather than running these scripts individually, we may decide to create a standard script that returns the common information we'll want to know. The second stage is to then run individual scripts to dig further into the issues. A good example of how to do this has been created by Alan Renouf (http://teckinfo.blogspot.com/2008/10/powershell-audit-script_15.html). The output from the WMI scripts is presented in an HTML page that can be viewed in a browser, as shown in figure 7.1

After discovering our information, we may need to make changes to the configuration. The machine configuration section finishes with some examples. Setting IP addresses is delayed until chapter 9.

It's also possible to work with what the user sees on the machine—his desktop experience. We can discover information about the desktop and other folders that are special to the user, including examining the contents of the Recycle Bin.

The final section of the chapter deals with applications. The Office applications Word and Excel have been chosen as examples because they can be found in most Windows environments. You can combine the discovery scripts presented in the first section with the information regarding Office applications to create a system to document and report on the machines in your environment. But before we can do that, we need to learn how to discover that information.

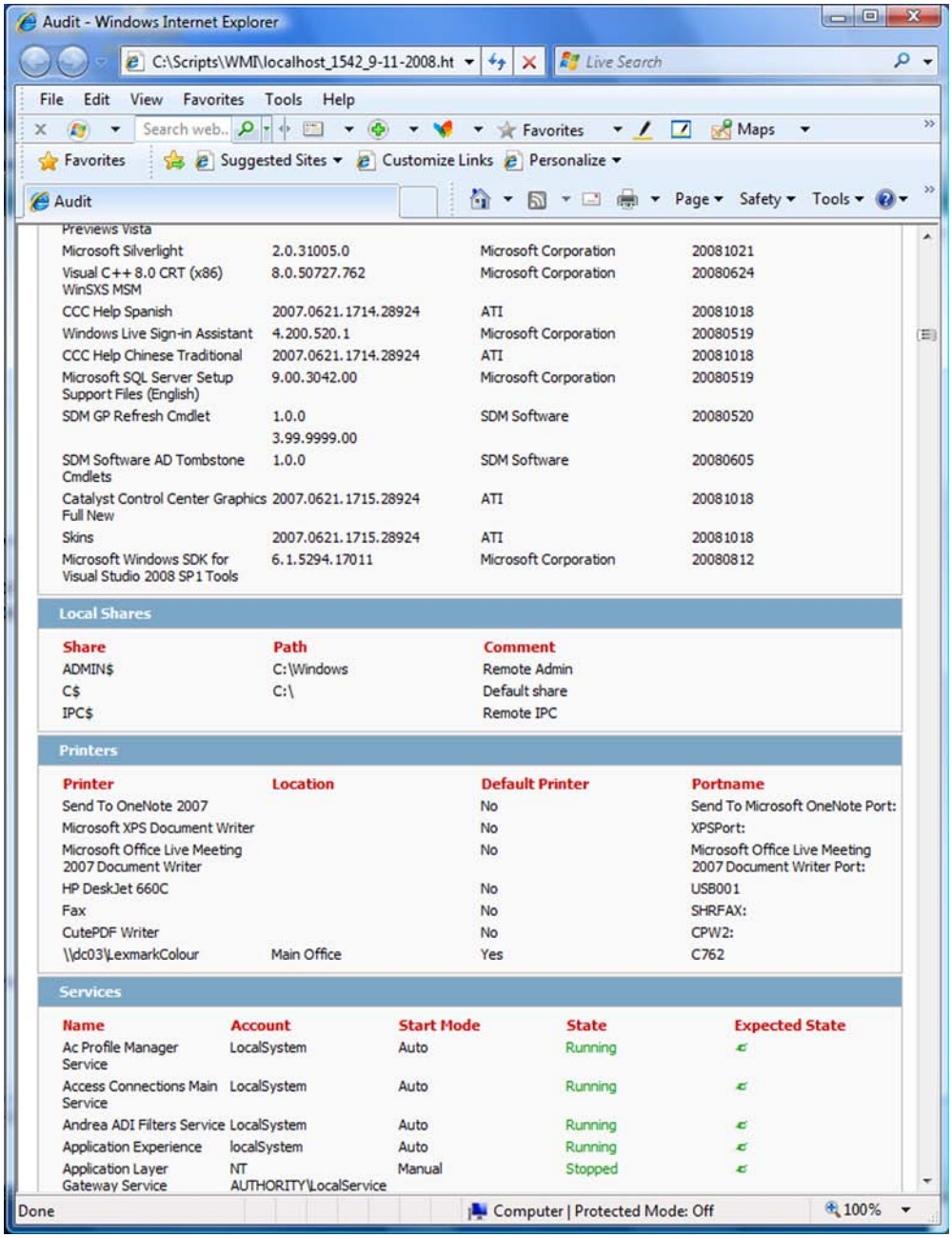


Figure 7.1 Presenting configuration information via HTML

7.2 Machine configuration

Machine configuration can be split into a number of areas. A lot of the time, when we need to work with a machine's configuration, we really just want to report on that

configuration. How many organizations have a database of their machine configurations? I suspect that the answer is not many. It would be relatively straightforward to take these scripts and push the results into a set of SQL Server tables to create such a database. We'll see an example in chapter 14.

> **REMOTE RUNNING** All of the scripts in this section are shown running against the local machine. They can be run equally well against a remote machine by using the `-ComputerName` parameter and supplying the NETBIOS name, IP address, or fully qualified domain name of the relevant computer. This doesn't require the remoting capabilities of PowerShell v2.

Examples of where we're only reporting include retrieving the OS or BIOS information. In some cases, we actually need to modify the machine configuration, for example altering the IP address or default gateway. This may be more likely on servers than workstations.

> **NOTE** Many of the scripts in this chapter use Windows Management Instrumentation (WMI) . WMI is blocked by default by the Windows firewall that's present in the latest versions of the Windows OS. Ensure that the firewall is configured to allow WMI access. WMI can take a long time to time out in certain instances. Ensure that firewalls aren't blocking WMI to avoid the wait.

WMI can supply a huge amount of information about your system. It's a constantly evolving technology, as each new version of Windows introduces new classes to the default namespace (root\cimv2) and also brings completely new namespaces. Most of the system information and machine configuration classes are contained in root\cimv2, as explained in the WMI section in chapter 3. You did read that section, didn't you? If you skipped that section, it would be worth reading to refresh yourself on how WMI works with PowerShell. One thing to remember with WMI is that you can use PowerShell itself to help you discover what classes are available by using the following:

```
Get-WmiObject -Namespace 'root\cimv2' -List
```

This will give you a list of the classes belonging to the namespace. Remember that the default namespace contains literally hundreds of classes. In order to reduce the wait, try limiting the number of classes returned by trying:

```
Get-WmiObject -Namespace 'root\cimv2' -List -Class *network*
```

Any suitable item can be used as the basis of the search. Wildcards are accepted as shown.

The logical place to start with the machine configuration is by obtaining some standard and fairly basic information about the machine. This is the sort of information that we're likely to need in many troubleshooting situations.

TECHNIQUE 40 System configuration

A system administrator, or help desk operative, needs to have access to system configuration information. Ideally this will be available to them through a configuration

database as described previously. If a configuration database isn't available then we can fall back on WMI to supply the information.

> **DATA SUBSET** This is only a small subset of the information that can be obtained. Examining the classes shown here or investigating other classes will enable a more detailed report to be generated.

The solution shown here is displayed onscreen. If a permanent record is required, `Out-File` could be used instead of `Format-List`. Remember to use the `-Append` parameter on all but the first section of the script; otherwise you'll experience one of those "Oops! I wish I hadn't done that" moments when you realize you've overwritten all of the data.

PROBLEM

The basic system configuration must be discovered. We want to know the OS and service pack, computer model and manufacturer, basic processor information, and the BIOS version.

SOLUTION

There isn't a single WMI class that we can use to retrieve all of the information we require. This means that a number of individual commands needs to be used, as shown in listing 7.1. The advantage of this approach is that it's easy to extend the script by introducing additional items such as physical disks or reporting on additional properties from the existing classes.

Listing 7.1 Basic system configuration information

```
"Operating System"                                          <-- 1 OS information
Get-WmiObject -Class Win32_OperatingSystem | Select Name,
Version,ServicePackMajorVersion, ServicePackMinorVersion,
Manufacturer, WindowsDirectory, Locale, FreePhysicalMemory,
TotalVirtualMemorySize, FreeVirtualMemory | Format-List

"Computer System"                                           <-- 2 Computer information
Get-WmiObject -Class Win32_ComputerSystem |
Select Name, Manufacturer, Model, CurrentTimeZone,
 TotalPhysicalmemory | Format-List

"Processor"                                                 <-- 3 Processor information
Get-WmiObject -Class Win32_Processor |
Select Architecture, Description | Format-List

"BIOS"                                                      <-- 4 BIOS information
Get-WmiObject -Class Win32_Bios |
Select -Property BuildNumber, CurrentLanguage,
InstallableLanguages, Manufacturer, Name,
PrimaryBIOS, ReleaseDate, SerialNumber,
SMBIOSBIOSVersion, SMBIOSMajorVersion,
SMBIOSMinorVersion, SMBIOSPresent, Status,
Version, BiosCharacteristics

Get-WmiObject -Class Win32_Bios |
Select ExpandProperty BiosCharacteristics                   <-- 5 Expand characteristics
```

WMI can supply a varied and rich set of information. In this example, we're drawing on four different WMI classes to supply the information. In all cases, the default namespace is being used (`root\cimv2`). In fact, it's used in all of the scripts in this section. The script is an ideal candidate for running against multiple computers.

We start by looking at the operating system ❶. As you might guess, we use the `Win32_OperatingSystem` class. Many classes in WMI are quite sensibly named, and so relatively easy to find. In this case, we select a number of properties, including the version, service pack, locale, and some information on the memory available in the machine. A `Format-List` is used at the end of the pipeline to ensure a consistent display.

> **NOTE** In all parts of the script, I could've dropped `Format-List` from the end of the pipeline and substituted it for the `select` command. That would be acceptable if I only wanted to display to the screen. The way I've written the script, I think it's easier to modify if you want to output the data to a file instead of to the screen. All you have to do is substitute `Out-File` and the name of the file in place of `Format-List`. Don't forget the `-append`.

`Win32_OperatingSystem` and the other classes used in this chapter have more properties than we're using in these examples. When we use `Get-WmiObject`, there's a default formatter that controls which properties are returned. The default formatters are investigated in appendix A. If you want to see the names of all of the properties and methods available to the particular WMI object we're working with, then use

```
Get-WmiObject -Class Win32_OperatingSystem | Get-Member
```

If you want to quickly see the values of the properties, use:

```
Get-WmiObject -Class Win32_OperatingSystem | Select *
```

The other point to note about this listing is that there's a label before each use of `Get-WmiObject`. In each case, we have a single string on the line. One convenient piece of functionality in PowerShell is that if we place a string by itself on a line, it's automatically treated as output to be directed to the standard output device—in this case the screen. This is good technique for listing progress through a script, as well as helping to comment the script. Two birds with one stone !

After the OS, the next area that will interest us is the machine itself ❷. This follows the same format as the previous line, except we're using the `Win32_ Computer-System class`. Information such as manufacturer, model, and the physical memory can be discovered. Troubleshooting a user's problem can often be easier and quicker when we know some basics about the machine. If we know that a particular model has had an issue with a piece of software, it could help narrow our search for a solution.

The final two sections of the script find basic information about the processor ❸ and the BIOS ❹.

SYSTEM DOCUMENTATION The possibility of using these scripts to populate a configuration database has been mentioned. The other possibility is to combine the scripts and output the results to a file. Any time there was a change to the machine, the script could be rerun. If required, the information could be written into a Word document using the scripts in the last section of this chapter as a guide.

WMI will sometimes return a collection of values as a single property. This happens with `Win32_Bios` in the `BiosCharacteristics` property ❺. We can use the `-ExpandProperty` parameter of `select` to list the members of the property collection.

DISCUSSION

In many cases, as here, the information in the property collection is stored as digits, each with a special meaning. Using the `-ExpandProperty` parameter doesn't add meaning; it just lists them. If you want to see what the numbers mean, the best way to find the information is to search on the Microsoft website for the documentation of the WMI class.

If it becomes necessary to add meaning to the numbers, pass the expanded characteristics into a `foreach` command that contains a `switch` statement. The meaning of each characteristic can be then be written out. I haven't included this, in the interest of space, but it's included in the downloadable scripts.

TECHNIQUE 41 Discovering the operating system

The operating system is fundamental to the correct operation of the machine and the installed software. As administrators, we need to know that the correct version is installed on our machines and that it's configured correctly.

PROBLEM

We need to be able to discover information about the version of the operating system and its configuration.

SOLUTION

WMI provides another class—`Win32_OperatingSystem`—to provide this information, as shown in listing 7.2. WMI has so much information available to us that we could almost start thinking "WMI is the answer; now what was my problem?"

Listing 7.2 Operating system information

```
Get-WmiObject -Class Win32_OperatingSystem |
Select BootDevice, BuildNumber,BuildType, Caption,
Codeset, CountryCode, Debug, InstallDate,
NumberofLicensedUsers, Organization, OSLanguage,
OSProductSuite, OSType, Primary, RegisteredUser,
SerialNumber, Version
```

DISCUSSION

The `Win32_OperatingSystem` class returns more than 70 properties. It's worth experimenting by using the following line of code in order to see the amount of information that can be retrieved:

```
Get-WmiObject -Class Win32_OperatingSystem | Select *
```

Several of these properties, such as `CountryCode` and `Codeset`, will be numbers. If required, their meanings can be discovered in the WMI documentation. It's usually enough to know what should be configured and then to look for the anomalies. After all, you'd expect all of the machines in the same location to use the same country code and OS language!

One issue that costs administrators a good deal of time and effort is patching. Knowing which service pack and hotfixes are installed on a machine is essential to keeping them secure.

TECHNIQUE 42 Discovering service packs on the OS

Service packs are issued at irregular intervals during the lifecycle of a version of Windows. A service pack will usually contain all of the previously released hotfixes and service packs in one package. Ideally, service packs should be installed as soon as they've been tested for your environment. Unless this is an automated procedure, it's easy for machines to be missed.

There are tools can tell us which service packs and hotfixes have been and still need to be installed on every system in our environment. When we're troubleshooting a problem, it's not always possible to access that information. Being able to generate a report showing the latest service pack installed on your machines could be very useful.

PROBLEM

We need to be able to discover the service pack applied to the operating system on a set of machines within our infrastructure.

SOLUTION

We can read a file to get the computer names and then discover the service pack level by using WMI. In previous examples, we've used `Import-Csv` to read a CSV file with the data we'll be using in the script. There's another cmdlet, `Get-Content` ❶, that we can use to read text files. As we're only reading a single column of names, either method will work.

> **SAMPLE FILE** A sample computers.txt file is supplied with the script downloads. It shows three different ways to call the local machine! Add other computer names or IP addresses as required.

It can be useful to read the contents of a file into an array, especially if you'll be using it multiple times in your script. Performing the read once will improve the performance and efficiency of your script. As an example, consider combining the scripts in listing 7.2 and listing 7.3. We're using the same data in both, so why read it twice?

Listing 7.3 Service pack information

```
$computers = Get-Content computers.txt
Foreach ($computer in $computers) {

    Get-WmiObject -ComputerName $Computer `
-Class Win32_OperatingSystem |
Select CSName, ServicePackMajorVersion,
ServicePackMinorVersion
    }
```

❶ Read file
❷ Loop through computers
❸ Get service pack version

Our data is in an array at this point, so we won't use the `Foreach-Object` cmdlet we've used in other scripts. We turn to the `foreach` loop command as shown ❷. Within the `foreach` loop, we use `Get-WmiObject` to access the `Win32_OperatingSystem` class ❸ to read the service pack version. One important change from using `Import-Csv` is that our `foreach` isn't on the pipeline. We can't use `$_` to represent the computer name in this instance. We use `$computer` instead, which is the `foreach` variable. Note that we're outputting the major and minor version numbers of the service pack. The minor version number will normally be zero.

Putting all of this together, the script can be read as:

1. Read a list of computer names.
2. For each computer name in the list.
3. Use WMI to get the service pack version.

A similar structure can be used to discover the hotfixes installed on a machine.

TECHNIQUE 43 Hotfixes

Hotfixes, or *patches*, are usually produced in response to a bug. They can be applied to a machine manually or by automatic means. It's possible that the method of application doesn't keep a centralized record of the patches that have been installed. Without this information, you, as an administrator, can't determine whether your machines are vulnerable to a new exploit that the patch will stop.

PROBLEM

We need to determine the hotfixes that are installed on a machine, and whether a particular hotfix is installed on the machine.

SOLUTION

We use a different WMI class this time. If this script is compared to the one in listing 7.4, we can see that they're very similar. We start, again, by reading the file of computer names ❶ and loop ❷ through that list. The output will consist of a list of the applied patches.

Listing 7.4 Hotfix information

```
$computers = Get-Content computers.txt

Foreach ($computer in $computers) {
 "`n "
  Get-WmiObject -ComputerName $Computer `
-Class Win32_QuickFixEngineering
}
```

❶ ❷ Loop through computers ❸ ❹ Get patch list

This could be quite long, so we put a blank line between the outputs of the different computers ❸. This is accomplished using the `` `n `` special character, which denotes a new line. Special characters are explained in appendix A.

We need to use a different WMI class to find the installed patches. We use `Win32_QuickFixEngineering` ❹. The default format has the information we need, so we don't need to use the `select` or `format` cmdlets.

POWERSHELL JOBS PowerShell v2 introduces the ability to run PowerShell commands as background jobs. This functionality enables us to perform tasks asynchronously. If we had a large number of computers in our list, it could take quite a while before all of the results were returned and we were given back control of the PowerShell prompt. This makes this type of task ideal for running as a job. When we run the command as a job, we get the prompt back immediately, so we can continue working while the job runs in the background. Once it has finished, we can investigate the results at our convenience.

DISCUSSION

In PowerShell v2, we could use a new cmdlet—`Get-Hotfix`—instead:

```
$computers = Get-Content computers.txt
Foreach ($computer in $computers) {
    "`n $computer"
    Get-HotFix -ComputerName $computer
}
```

As it stands, the script will display all of the hotfixes installed on the system. Much of the time, we may want to check whether a particular hotfix is applied on the machines, as in listing 7.5.

Listing 7.5 Check for specific hotfix

```
$computers = Get-Content computers.txt
Foreach ($computer in $computers) {
$fix = Get-WmiObject -ComputerName $computer
-Class Win32_QuickFixEngineering -Filter "HotfixId = 'KB998989'"
if ($fix -eq $null){Write-Host "$computer - patch not installed"}
else {Write-Host "$computer - patch installed"}
}
```

❶ Read file
❷ Loop through computers
❸ Get hotfix
❹ Print message

This script starts in the same way, by reading the file of computer names ❶ and looping through them ❷. In this case, we put the results ❸ of the `Get-WmiObject` cmdlet into a variable. The `-Filter` parameter is used to restrict `Get-WmiObject` to only look for the specified hotfix. This will give a significant boost to the script's performance. The way PowerShell works is that it will return nothing if it can't find the specified patch on the machine. We can exploit that fact to determine the patch's existence.

If the patch doesn't exist the variable is null (nothing has been returned). We can use an `if` statement ❹ to test whether the variable is null, and if it is, we get a message to tell us the patch isn't installed. If the variable isn't null it means that WMI found the patch and we can write out the appropriate message.

`Get-Hotfix` can also be used in this situation:

```
$computers = Get-Content computers.txt
Foreach ($computer in $computers) {
    "`n $computer"
    Get-HotFix -Id KB972636 –ComputerName $computer
}
```

In addition to the installed hotfixes, we'll also be interested in the installed software.

TECHNIQUE 44 Listing installed software

Many organizations will have policies to determine what software is allowed to be loaded on desktop machines. If a software asset management system isn't in place, it can be difficult to know what software has been installed. Also, when troubleshooting, it can be useful to know what software has been installed, and especially what versions of the software.

PROBLEM

We need to create a report listing the software loaded on our machines.

SOLUTION

The `Win32_Product` WMI class is available on all recent versions of Windows, up to and including Windows Server 2008, Windows Vista, and Windows 7. We use the `Win32_product` class to read the list of software installed under the control of the Windows installer, as shown in listing 7.6.

Listing 7.6 Installed software

```
Get-WmiObject -Class Win32_product | Select Name, Caption,
IdentifyingNumber, InstallLocation, Vendor, Version |
 Export-Csv software.txt -NoTypeInformation
```

DISCUSSION

Within the PowerShell community, a lot is made of the "PowerShell one liner." The way the pipeline works in PowerShell enables us to put a lot of functionality into a single line of code. This script, though fairly simple, achieves a lot. It's also expandable, while remaining (technically) within the one-line constraint.

> **IMPORTANT NOTE** Only software installed through the Windows installer will be discovered by this script. This will include most modern software, but there's always the possibility that some software has been installed by other means.

The script returns a lot of information, as with most WMI classes. A select (`Select-Object` cmdlet) is used to restrict the data passed along the pipeline. Our problem statement was that we needed to create a report listing the installed software.

One of the easiest ways to do this is to create a CSV file containing the information. We can use the `Export-Csv` cmdlet to create and write to the CSV file. By default. `Export-Csv` will write the .NET type information into the first row of the CSV file. This can cause problems when we try to read the CSV file outside of PowerShell. The `-NoTypeInformation` parameter will prevent the type information being written to the file.

The example only creates a report for the local machine. We can extend the script to work with remote machines. Create a file with the computer names. This can be a CSV file or a text file. Read the file and pipe it into a `foreach` loop. Use the `-Computername` parameter in `Get-WmiObject` to read the information from the appropriate computer. The information can then be exported to a file. In these cases, we usually want to create

one file per computer, so create a filename in the loop and incorporate the computer name and the date.

Our investigation of the machine configuration has encompassed the hardware, the OS, and the installed software. This gives us the basic building blocks, but one area we haven't looked at yet is the disks.

TECHNIQUE 45 Monitoring free disk space

As you'd expect, there are a number of classes that deal with disks:

- `Win32_DiskDrive`
- `Win32_DiskDrivePhysicalMedia`
- `Win32_DiskDriveToDiskPartition`
- `Win32_DiskPartition`
- `Win32_DiskQuota`
- `Win32_LogicalDisk`
- `Win32_LogicalDiskRootDirectory`
- `Win32_LogicalDiskToPartition`
- `Win32_LogonSessionMappedDisk`
- `Win32_MappedLogicalDisk`

Physical disks are investigated using `Win32_DiskDrive`. `Win32_DiskDriveToDiskPartition` is especially useful, as it enables us to discover which partitions are on which physical disk. In this instance, we're concerned with disk space.

PROBLEM

Monitoring available disk space is a common task for administrators. We may also need to quickly discover how much free space is available on a given disk when we need to move data around or install software.

SOLUTION

If we use the `Win32_LogicalDisk` class and a calculated field, we can even get the answer in GB rather than bytes, as in listing 7.7.

Listing 7.7 Free disk space

```
$HardDisk = 3
Get-WmiObject -Class Win32_LogicalDisk `
-Filter "DriveType = $HardDisk" |
Format-Table DeviceId, @{Label="Freespace(GB)";
Expression={($_.FreeSpace/1GB).ToString("F04")}} -auto
```

There are a number of interesting points in this script. In the first line of the script, I've defined a variable called `$HardDisk` with a value of 3. This is then used in the `-Filter` parameter of `Get-WmiObject`. A filter is in effect a WMI Query Language (WQL) query, but we only need to give the part of the query after the `WHERE` statement. The full query would be:

```
"SELECT * FROM Win32_LogicalDisk WHERE DriveType = '$HardDisk'"
```

The WHERE clause is acting as a filter, and so is the part we use in our filter parameter as shown in the script. In listing 7.7, we've defined the query to only look at disks where the drive type equals the value of the variable.

The query will only return information on local hard disks—type 3 disks. There's a clue in the name of the variable. I could've just hard-coded the disk type into the script, but this allows us to change the disk type easily, as it's at the top of the script. It also demonstrates how we can substitute into strings. If a variable is placed inside a double-quoted string as in this example, its value will be substituted into the string when the script executes.

> **QUOTES** The way that PowerShell allows us to put variables inside strings and then substitute the value of the variable is a useful technique. It greatly simplifies a number of tasks, especially building commands in this way and creating output messages. It only works with strings bounded by double quotes.

DISCUSSION

The second interesting—and again useful—feature is the calculated field in the `Format-Table` cmdlet. It reads:

```
@{Label="Freespace(GB)"; Expression={($_.FreeSpace/1GB).ToString("F04")}}
```

This isn't as complicated as it would seem, as we can see by breaking this down. Start at the outside with `@{}`, which denotes this is a hash table. We know that hash tables consist of key-value pairs. In this case, we have two keys—`Label` and `Expression`. The `Label` is a string that defines the name of the field, as shown in figure 7.2.

The `Expression` is where all the hard work is done, as we take the value of the `Freespace` property (which is in bytes) and convert it to gigabytes by dividing by 1GB. Remember that PowerShell recognizes KB, MB, and GB as values. This calculation leaves a large number of decimal places, so we'll format the output by converting the number to a string using `"F04"` as the format string. This will restrict the display to four decimal places. What we've done is take a value in bytes and convert it into a more meaningful form given the size of the hard disks in use today.

Calculated fields can be used in a `select` statement, except that `Label` is changed to `Name`:

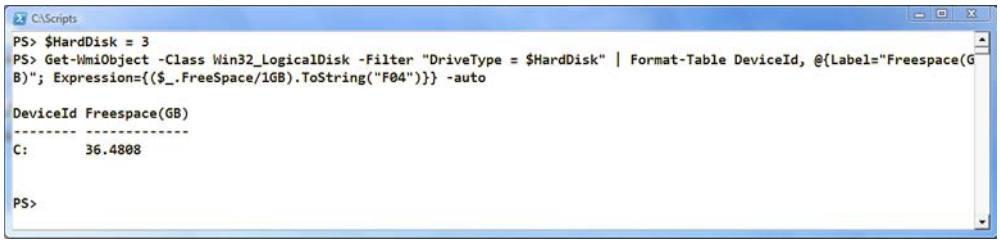


Figure 7.2 Using a calculated field

```
$HardDisk = 3
Get-WmiObject -Class Win32_LogicalDisk `
-Filter "DriveType = $HardDisk" |
Select DeviceId, @{Name="Freespace(GB)";
Expression={($_.FreeSpace/1GB).ToString("F04")}}
```

When we create a calculated field using `select`, it becomes part of the object and can be used later on the pipeline exactly like any other property. The member type for these properties created as calculated fields is `NoteProperty`.

> **GET-PSDRIVE** In PowerShell v2, `Get-PSDrive` has been amended to give used and free space for filesystem drives.

One configuration item that can have a major impact is the IP address. We'll look at that in chapter 9.

TECHNIQUE 46 Renaming a computer

When a Windows machine is built, it'll be assigned a randomly created name. This name will need to be changed to fit our naming conventions. We may also need to change the name of a machine if it changes role or location.

PROBLEM

How can we change the name of a Windows machine?

SOLUTION

Another aspect of the `Win32_ComputerSystem` class comes into play, as in listing 7.8.

Listing 7.8 Change computer name

```
$computer = Get-WmiObject -Class Win32_ComputerSystem
$computer.Rename("newname",$null,$null)
```

Set a variable to `Win32_ComputerSystem`. The `Rename()` method is used to change the name. We need to give the new computer name as the first parameter. The other parameters are the administrator ID and password, which aren't necessary if you're working on the local machine and are logged in as the administrator.

DISCUSSION

There are a couple of points worth remembering about computer renaming. Using this technique on a machine does *not* change the name in Active Directory. If the machine is a domain member, perform the name change through the system configuration to ensure AD is updated.

In the PowerShell v2 beta process, a `Rename-Computer` cmdlet was introduced. This was removed in the final version. Any mention of using this cmdlet you find in documentation or examples should be ignored.

Changing the name of the computer will force a reboot. There are also times when we need to manually reboot or shut down the system.

TECHNIQUE 47 Restarting a computer

There are many situations when we need to restart or even shut down a system. The ability to restart a system remotely is useful. If we have a remote desktop connection,

we can log on and perform the reboot. It's more efficient to run the first option in this script.

There are many applications that use a number of different servers. These systems must be shut down in the right order; for example if we have a SharePoint environment, we must shut down the SharePoint servers before we shut down the SQL Server back end. If we perform the shutdowns by a script, we'll always get them in the right order. This is important if we're working late and need to get it right!

PROBLEM

What's the most efficient way to remotely restart or shut down a system?

SOLUTION

The OS controls the machine, so we need to look at the `Win32_OperatingSystem` class.

> **WHATIF WARNING** There's no warning, `whatif`, or confirmation when using these WMI methods. If we use the PowerShell 2.0 `Restart-Computer` cmdlet, we can use the `-whatif` parameter. Use `Stop-Computer` for a complete shutdown.

In listing 7.9, we start by creating a variable to represent the computer's `Win32_OperatingSystem` class. We can then call the `Reboot()` method ❶ to cause a restart, or we can make the machine shut down by using `Win32Shutdown()` ❷. In this example, a computer name is given, so it's acting on a remote machine.

Listing 7.9 Restarting the system

```
$computer = Get-WmiObject -CompterName pcrs2 -Class Win32_OperatingSystem
$computer.ReBoot()                                          ❶ Reboot

$computer = Get-WmiObject -CompterName pcrs2 -Class Win32_OperatingSystem
$computer.Win32Shutdown(5)                                  ❷ Shutdown
```

DISCUSSION

The numeric value we pass as a parameter to `Win32Shutdown()` defines how the machine closes down. Table 7.1 shows that the values are paired with a difference of four between the members of a pair. The second value in the pair causes the activity to happen immediately and force the closure of any open applications.

We've seen in this section that we can discover a lot of extremely valuable information about the configuration of our systems using PowerShell and WMI. We can also configure the machines remotely using the same tools. These activities affect the machine, but what can we do for the user? This aspect of the desktop will be examined next.

Value	Meaning
0	Log off
4	Forced logoff
1	Shutdown
5	Forced shutdown
2	Reboot
6	Forced reboot
8	Power off
12	Forced power off

Table 7.1 `Win32Shutdown` values

7.3 User features

In an Active Directory-based environment, a lot of the configuration work that directly affects the user will be performed by Group Policy. PowerShell isn't directly usable from Group Policy. There's a certain amount of configuration work we may need to perform, especially concerning the folders such as the desktop that are known as *special folders*. Printers and the recycle bin are parts of the desktop environment that we may need to work with when administering the user's desktop.

See appendix D for a list of special folders.

As an introduction to special folders, we'll work with the desktop folder.

TECHNIQUE 48 Minimizing windows

When working on a Windows machine, we often find ourselves in the position of having multiple windows open. Many years ago when I spent my time directly supporting users, I remember being called over by a user who was having difficulty opening a particular spreadsheet. After a bit of digging, we realized he already had the file open but couldn't see it because it was hidden behind a number of other windows. A function to minimize all of the open windows would've been useful.

PROBLEM

I have too many windows open on my desktop and have lost track of what's open. I don't want to close the applications, as I have work in progress using the applications.

SOLUTION

We can use the `Shell` object to help us in this situation, as in listing 7.10.

Listing 7.10 Minimizing desktop windows

```
$a = New-Object -ComObject Shell.Application
$a.MinimizeAll()
```

The shell is the interface we work with in Windows OSs. The `Shell` COM object gives us the ability to access and work with that shell. We can create an object representing the shell by using `New-Object`. We need to use `-ComObject`, as this is COM-based. If we don't put in this parameter, PowerShell will assume that we're trying to work with a .NET object that it won't be able to find. This will cause it to object (sorry) by throwing an error.

DISCUSSION

There are a number of useful methods on the `Shell` object, which we can find by using:

```
$a | Get-Member
```

We can stop and start services, set the machine time, shut down Windows, and search for files and printers, for example. Using the `MinimizeAll()` method, as shown in the listing `Get-Member` produces, causes all of the windows to minimize. It's possible to use `UndoMinimizeAll()` to reverse this.

Now we've minimized all of the windows we can see the desktop. Users seem to fall into two groups—those who have hardly any icons on their desktops and those whose desktop is completely covered in icons. There's a problem with this last situation.

TECHNIQUE 49 Desktop contents

The problem with having a lot of icons on the desktop is that only a few characters of the file or application name are displayed. This can lead to a lot of wasted time as we search through the icons on the desktop looking for that elusive file.

PROBLEM

I have too many files and icons on my desktop. How can I see what's really there?

SOLUTION

Using the `Shell` object, we can drill down into the desktop contents, as in listing 7.11.

Listing 7.11 Viewing the desktop contents

```
$a = New-Object -ComObject Shell.Application
$desktop = 0x0
Get-ChildItem $a.Namespace($desktop).Self.Path
```

DISCUSSION

We start as before, by creating an instance of the `Shell` object. We need to set a variable to represent the desktop. This is in hexadecimal format, as shown by the `0x` prefix. Appendix B lists the special folders and their representative values. A lot of the scripts that you see on the web will have the values for the special folders shown as hexadecimal.

`Get-ChildItem` will return a list of the files in the desktop folder. We specify the path to the desktop folder as shown.

Hexadecimal conversions

We can use the following function to convert a decimal number to hexadecimal:

```
Function tohex{
param ([int]$i =0)
[convert]::ToString($i,16)
}
```

This function accepts an integer as input and uses the `System.Convert` class to convert the integer to a string represent a hexadecimal (base 16) number. This method can also be used to convert to binary (base 2) and octal (base 8). If you need to convert a hexadecimal number back to a decimal number, use this conversion: `[convert]::ToString(0xF,10)`. The number to be converted has to be presented in hexadecimal format.

TECHNIQUE 50 Adding a file to the desktop

I store a number of small files on my desktop. Usually they're pieces of information that I know I'll require on a frequent basis, such as the IP addressing scheme for my virtualized environment (don't ask) or important information such as the deadline for my next chapter that I have to keep handy. Opening Notepad, typing the information, and saving the file to the desktop is tedious. There's an easier way to do it.

PROBLEM

We want to preserve the current process information in a file on the desktop so that we can refer to it at a later time.

SOLUTION

This problem can be solved by modifying listing 7.11. We create the COM object representing the shell. We can then use the `desktop` namespace to create a file path, as in listing 7.12.

Listing 7.12 Create a file on the desktop

```
$a = New-Object -ComObject Shell.Application
$file = $a.Namespace(0x0).Self.Path + "\process.csv"
Get-Process | Export-Csv -Path $file-NoTypeInformation
```

`Get-Process` can be piped into an `Export-Csv` that writes the data into our file. `-NoTypeInformation` prevents the .NET type information being written to the first line of the CSV file

DISCUSSION

We could create a string holding some data and pass that to `Out-File` to create a TXT file on the desktop instead. If we need to access the file, we can create the file path as shown.

TECHNIQUE 51 Listing cookies

Many internet sites will create a cookie to hold information relevant to your visit to the site. The problem with cookies is that you don't necessarily know that they've been created. In the past, I've had problems with particular sites that changed in various aspects. The cookies I had for the site wouldn't work with the new version, so I had to delete the cookies. Internet Explorer is an all-or-nothing proposition. It'd be better to be able to find which cookies were causing the problem and delete only those, rather than all cookies.

PROBLEM

What cookies have been saved on your machine? You don't know?

SOLUTION

We can access the cookie folder (don't you wish they'd been called jars rather than folders?) and determine which cookies are present, as in listing 7.13.

Listing 7.13 Discovering cookies

```
$a = New-Object –ComObject Shell.Application
Get-ChildItem $a.Namespace(0x21).Self.Path  |
Sort LastWriteTime -Descending
```

The shell object is created and the `Get-ChildItem` cmdlet is used to list the contents of the cookie folder. We can sort the information using the PowerShell pipeline to pass the objects into `Sort` based on `LastWriteTime`.

DISCUSSION

This means we can easily see the most recent cookies. It would be a simple matter to replace `Sort LastWriteTime –Descending` with

```
Where {$_.LastWriteTime -le (Get-Date).AddDays(-90)} | Remove-Item
```

This would enable us to delete old cookies. Set the number of days to compare against to a value that suits your system.

Another special folder we need to consider is the recycle bin.

TECHNIQUE 52 Viewing recycle bin contents

When we delete a local file from our machines, it doesn't immediately vanish. It's moved into the recycle bin for possible restoration. We need to empty the recycle bin to finally remove the file.

> **ACKNOWLEDGMENT** This script and the following one are adapted from a blog post by Thomas Lee: http://tfl09.blogspot.com/2007/01/manipulating-recycle-bin-in-powershell.html. These scripts don't work on Windows Vista/Windows Server 2008 and above, but alternatives are provided.

PROBLEM

It looks like an important file we need has been deleted. How can we check the recycle bin to see if it's there?

SOLUTION

The recycle bin is a special folder and can be accessed in the same way as the others, as in listing 7.14.

Listing 7.14 Opening the recycle bin

```
$a = New-Object -ComObject Shell.Application
$rb = $a.NameSpace(0xa)
$rb.Self.InvokeVerbEx("Explore")
```

❶ Recycle bin namespace

❷ Open bin

We create the shell object as before, but this time we create a variable representing the recycle bin namespace ❶. `Self` represents the recycle bin folder. We use the `InvokeVerb()` method to explore (open) the recycle bin window ❷. This would make a useful function that could be invoked with a single command.

DISCUSSION

The recycle bin folder can be investigated using our trusty `Get-Member` cmdlet:

```
$rb.Self | gm
```

`Get-Member` has `gm` as an alias.

ALIASES I find I spend more time writing scripts than working interactively. I only use aliases at the command prompt, and even then only the most common ones, as this ensures my scripts are understandable and portable. I recommend that custom aliases never be used in scripts. They may not be present on other machines.

If we dig a little further, we find a `verbs()` method:

```
$rb.self.verbs()
```

The verbs in the list match the context menu you get when you right-click a desktop object. We could use this method to open a file or application on the desktop. If you're using Windows Vista/Windows Server 2008 or above, this script may not work. In that case, we can use plan B and access the recycle bin contents like this:

```
$a = New-Object -ComObject Shell.Application
$a.NameSpace(0xa).Items() | Format-Table Name, Path -AutoSize
```

The recycle bin has a verb that we can use to empty it, as we'll see next.

TECHNIQUE 53 Emptying the recycle bin

The contents of the recycle bin can take up valuable disk space. We can reclaim this space by emptying the recycle bin.

WARNING Once a file has been deleted from the recycle bin, there's no native way to restore that file. It means turning to our backup system. You do perform backups, right?

PROBLEM

It would be useful to empty the recycle bin on a periodic basis as part of our housekeeping routines for a machine. Can we do that from a script?

SOLUTION

The recycle bin has an entry on its context menu to empty it. We gain access to the recycle bin as before. We then use `InvokeVerb()` to trigger the action, as in listing 7.15. A confirmation dialog box will pop up asking if you want to perform the action.

Listing 7.15 Emptying the recycle bin

```
$a = New-Object –ComObject Shell.Application
$rb = $a.NameSpace(0xa).Self
$rb.InvokeVerb("Empty Recycle &Bin")
```

DISCUSSION

This technique works on Windows XP/2003 but not on Windows Vista/2008 or later. An alternative for Vista, and above, would be to use this to pipe the items in the recycle bin into `Remove-Item`:

```
$a =  New-Object -Com Shell.Application
$a.NameSpace(0xA).Items() | Remove-Item -Recurse
```

TECHNIQUE 54 Sending a printer test page

One thing that sticks in my mind from my time directly supporting users is that a lot of reported problems revolved around printing. My colleagues working in this area assure

me that this still is the case. I dedicate the next couple of scripts to every administrator who has had to troubleshoot a printing issue with the hope they may be of some help.

PROBLEM

One step that's often needed when we start investigating printing issues is to attempt to print a test page. This will quickly show whether we can communicate with the printer. Usually we do this through the GUI controls, or we ask the user to do it on a remote machine. Can we perform this action remotely?

SOLUTION

We've seen that WMI has the capability to access remote machines. There's a WMI class specifically meant for working with printers, as shown in listing 7.16.

Listing 7.16 Print a test page

```
Get-WmiObject -Class Win32_Printer | Select Name      ❶
$printer = Get-WmiObject -Class Win32_Printer      ❷
-Filter "Name = 'HP DeskJet 660C'"
$printer.PrintTestPage()            ❸
```

We need to start by checking the printers installed on the machine ❶. This enables us to ensure that we're working with the correct printer. The `-computername` parameter can be used to specify a remote machine to access. Once we know the name of the printer we need, we can use that in a filter ❷ to create an object for the printer. A call to the `PrintTestPage()` method produces the test page ❸.

> **PRIVILEGES** When using Windows Vista/2008, we need to start PowerShell with elevated privileges. We right-click the PowerShell icon and select Run As Administrator.

DISCUSSION

We could test all printers on a machine using a `foreach-object` cmdlet and combining the lines of code. Alternatively listing 7.16 could be used to test a printer on a number of machines. The return code would have to be checked to confirm the test page was successfully printed.

The other aspect of printers that we need to consider is the printer drivers that are installed on the machine.

TECHNIQUE 55 Printer drivers

Printer drivers can cause issues. I've seen drivers for printers from the same manufacturer causing problems because they used different versions of the same file. We need to be able to determine information about our printer drivers.

PROBLEM

We have a number of printers installed on the machine; how can we check that the correct drivers are installed?

SOLUTION

WMI can provide this information—specifically the `Win32_PrinterDriver` class. A `Select` statement will reduce the amount of data returned to us, as in listing 7.17. Use `Get-Member` to determine other useful information that's available.

Listing 7.17 View printer drivers

```
Get-WmiObject -Class Win32_PrinterDriver |
Select Name, ConFigFile, DependentFiles, Driverpath |
Format-List
```

DISCUSSION

We've examined the machine configuration and how we can work with features of interest to the user such as the desktop and their printers. The last section in this chapter will cover working with the standard Office applications Word and Excel.

7.4 Office applications

It's a fair assumption to say that the Microsoft Office applications will be found on almost every desktop machine in work environments. It's possible to work with most of the Office applications using PowerShell. There are COM objects representing most of the them. In this section, we'll concentrate on using Word and Excel, as these are the two applications we're most likely to use as administrators. We'll start with Excel by looking at how we can create a spreadsheet and then put data into it. Spreadsheets seem much more useful when they have data. In the same way, we'll look at creating a Word document and how to push text into the document.

> **NOTE** In this section, we'll just look at using the COM methods to work with Excel and Word. If the Office 2007 applications are in use, it's possible to use the OpenXML format for the documents. This involves delving into the depths of XML, which may not appeal to all administrators. An example of using OpenXML will be given at the end of this section.

The Microsoft Technet script center has a lot of VBScript examples of using Excel that can be converted to PowerShell. The first thing we need to do is to create an Excel spreadsheet.

TECHNIQUE 56 Creating an Excel spreadsheet

Creating an Excel spreadsheet should be a simple act, in theory. But there's a slight problem in the shape of a bug in Excel versions up to Excel 2007 that can prevent this from working if you don't happen to be in the U.S. After reading this, it won't matter where you live. If you're using Excel 2010, then the first version in listing 7.18 can be used wherever you live and work.

PROBLEM

We need to create an Excel spreadsheet from within a PowerShell script.

SOLUTION

The `Excel.application` COM object can be used to create a spreadsheet.

Listing 7.18 Create Excel spreadsheet

```
$xl = New-Object -ComObject "Excel.Application"    <-- (1) U.S. version
$xl.visible = $true
$xlbooks =$xl.workbooks.Add()
```

```
$xl = New-Object -ComObject "Excel.Application"
$xl.visible = $true
$xlbooks =$xl.workbooks
$newci = [System.Globalization.CultureInfo]"en-US"
$xlbooks.PSBase.GetType().InvokeMember("Add",
[Reflection.BindingFlags]::
InvokeMethod, $null, $xlbooks, $null, $newci)
```

❷ **International version**

DISCUSSION

If you live in the U.S. or are using a machine that's configured to the U.S. locale—see the Control Panel -> Regional and Language settings (figure 7.3). Then you can use the first option in listing 7.18.

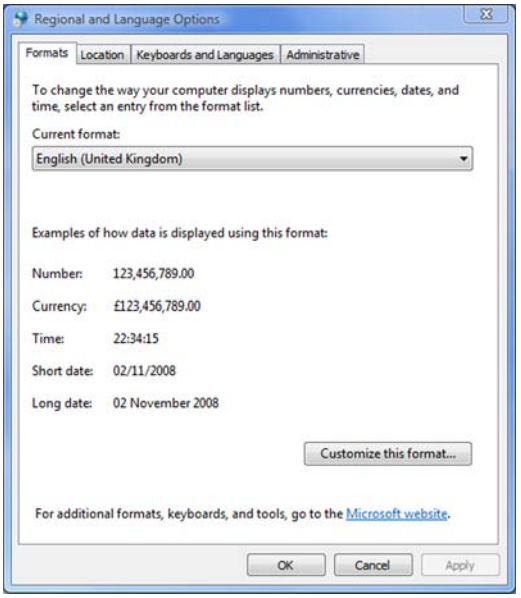


Figure 7.3 Regional settings

Otherwise, you have to use the second, international option. If you want to remain with PowerShell rather than succumbing to the GUI, you can check the culture by typing `$psculture` (in PowerShell v2). If `en-US` isn't returned then you need to use the second option in listing 7.18.

The simple way to create a spreadsheet ❶ starts by creating the COM object using `New-Object`. We make it visible. Administrators are clever people, but working on an invisible spreadsheet may be a step too far, especially on a Monday morning. At this point, we have only the Excel application open. We need to add a workbook to enable us to use the spreadsheet.

If the machine isn't using the U.S. culture—I live in England so `$psculture` returns `en-GB`—we have two options. The first option is to change the culture on the machine to `en-US`, which isn't convenient. Otherwise, we have to use the second option given in the listing.

We start in the same way by creating the COM object and making the spreadsheet visible. A variable `$xlbooks` is created that represents the workbooks in the spreadsheet. A second variable `$newci` is created that represents the culture. Note that we're forcing the culture used to create the workbook to be U.S. English. The last line is a bit complicated, but we're dropping down into the base workbook object and invoking the `add` method using the U.S. English culture. If you don't want to see the long list of data onscreen when this last line is run, then add `| Out-Null` to the end of the line. This is awkward, but it does get us past the bug. The good news is that once we've created our workbook, we can add data into it.

TECHNIQUE 57 Adding data to a spreadsheet

A spreadsheet without data isn't much use to us, so we need to investigate how we can add data into the spreadsheet and perform calculations on that data.

PROBLEM

We need to populate our spreadsheet with some data.

SOLUTION

Expanding on the previous script, we can create a worksheet to hold the data. The starting point is to remove any previous versions of the spreadsheet ❶, as shown in listing 7.19. We use `Test-Path` to determine whether the file exists and `Remove-Item` to delete it. The `-confirm` parameter could be used with `Remove-Item` as an additional check if required. This is useful if working with important data.

Listing 7.19 Add data to Excel spreadsheet

```
$sfile = "C:\test\test.xlsx"
if(Test-Path $sfile){Remove-Item $sfile}

$xl = New-Object -comobject "Excel.Application"
$xl.visible = $true
$xlbooks =$xl.workbooks
$newci = [System.Globalization.CultureInfo]"en-US"
$wkbk = $xlbooks.PSBase.GetType().InvokeMember("Add",
[Reflection.BindingFlags]
::InvokeMethod, $null, $xlbooks, $null, $newci)
$sheet = $wkbk.WorkSheets.Item(1)

$sheet.Cells.Item(1,1).FormulaLocal = "Value"
$sheet.Cells.Item(1,2).FormulaLocal = "Square"
$sheet.Cells.Item(1,3).FormulaLocal = "Cube"
$sheet.Cells.Item(1,4).FormulaLocal = "Delta"

$row = 2

for ($i=1;$i -lt 25; $i++){

    $f = $i*$i

    $sheet.Cells.Item($row,1).FormulaLocal = $i
    $sheet.Cells.Item($row,2).FormulaLocal = $f
    $sheet.Cells.Item($row,3).FormulaLocal = $f*$i
    $sheet.Cells.Item($row,4).FormulaR1C1Local = "=RC[-1]-RC[-2]"

    $row++
}

 [void]$wkbk.PSBase.GetType().InvokeMember("SaveAs",
[Reflection.BindingFlags]
::InvokeMethod, $null, $wkbk, $sfile, $newci)

[void]$wkbk.PSBase.GetType().InvokeMember("Close",
[Reflection.BindingFlags]
::InvokeMethod, $null, $wkbk, 0, $newci)
$xl.Quit()
```

❶ Delete previous files
❷ Create spreadsheet
❸ Set headers
❹ Row counter
❺ Create data
❻ Save
❼ Close
❽ Quit

The next step is to create the spreadsheet. In this case, I've used the international method. Once the workbook is created, we can create a worksheet ❷. Worksheet

cells are referred to by the row and column as shown ❸ by creating the column headers.

A counter is created ❹ for the rows. A `for` loop ❺ is used to calculate the square and the cube of the loop index. This is a simple example to illustrate the point. In reality, the data could be something like the number of rows exported compared to the number of rows imported for each table involved in a database migration. Note that the difference between the square and the cube is calculated by counting back from the current column.

We save the spreadsheet when all of the data has been written to it ❻ and close the workbook ❼. Note that we have to use a similar construction to adding a workbook, in Excel 2007 and earlier, to get around the culture issue. If we were using the `en-US` culture, those lines would become:

```
$wkbk.SaveAs("$file")
$wkbk.Close()
```

The last action is to quit the application ❽.

DISCUSSION

There are numerous reasons why you would want to record data into a spreadsheet but the performance implications must be understood. Working with Excel in this manner can be slow.

> **RECOMMENDATION** Adding data into an Excel spreadsheet in this manner can be extremely slow. In fact, painfully slow if a lot of data needs to be written into the spreadsheet. I strongly recommend creating a CSV file with the data and manually importing it into Excel instead of working directly with the spreadsheet.

This technique could be used to create reports, for instance from some of the WMI-based scripts we saw earlier. The machine name and relevant information could be written into the spreadsheet. Alternatively, we can write the data to a CSV file and then open it in Excel.

TECHNIQUE 58 Opening a CSV file in Excel

We have seen how writing data directly into a spreadsheet is slow. Slow tends to get frustrating, so we need another way to get the data into a spreadsheet. If we can write the data to a CSV file, we can open that file in Excel. It's much faster and more satisfying.

PROBLEM

Having decided that we need to speed up creating our spreadsheet, we need to open a CSV file in Excel.

SOLUTION

The `Open` method will perform this action, as in listing 7.20.

Listing 7.20 Open a CSV file

```
$xl = New-Object -comobject "excel.application"
$xl.WorkBooks.Open("C:\Scripts\Office\data.csv")
$xl.visible = $true
```

DISCUSSION

As with previous examples, we start by creating an object to represent the Excel application. We can then use the `Open` method of the workbooks to open the CSV file. The only parameter required is the path to the file. The full path has to be given. We then make the spreadsheet visible so we can work with it.

Alternatively we could use:

```
Invoke-Item data.csv
```

This depends on the default action in the file associations being to open the file in Excel. Hal Rottenberg graciously reminded me of this one.

The other major Office application is Word, which we'll look at next.

TECHNIQUE 59 Creating and writing to a Word document

Creating a Word document is straightforward compared to creating an Excel spreadsheet. There's only a single method regardless of location. The ability to add text into a Word document from within our script enables us to automate our processes and create the reports detailing our activities all in one pass. This is efficiency.

PROBLEM

We need to create a report from within our script.

SOLUTION

We adapt the script from listing 7.18 to create the report. The `New-Object` cmdlet ❶ is used to create an instance of the `Word.Application` object, as shown in listing 7.21. Note that we need to tell PowerShell we're dealing with a COM object. If this is forgotten, the error message will say that it can't find the type `word.application` and tell you to check that the assembly is loaded.

Listing 7.21 Add text to a Word document

```
$word = New-Object -ComObject "Word.application"
$word.visible = $true
$doc = $word.Documents.Add()
$doc.Activate()

$word.Selection.Font.Name = "Cambria"
$word.Selection.Font.Size = "20"
$word.Selection.TypeText("PowerShell")
$word.Selection.TypeParagraph()

$word.Selection.Font.Name = "Calibri"
$word.Selection.Font.Size = "12"
$word.Selection.TypeText("The best scripting language in the world!")
$word.Selection.TypeParagraph()

$file = "c:\scripts\office\test.doc"
$doc.SaveAs([REF]$file)
$Word.Quit()
```

❶ Create Word document

❷ Add header

❸ Add text

❹ Save document

❺ Quit Word

The Word application is made visible so that we can work with it. Finally, we add a document to the application. At this point, we have a blank document to start typing into, exactly as if we'd double-clicked on the Word icon. We can take this a stage further by writing text into the document from our script.

A header can be added ❷ by defining a font name, a font size, and the text to be added. The `TypeParagraph()` method is used to denote a new paragraph. These actions are repeated ❸ to add text to the document. The script closes by defining the filename to be used by the file; we save the file ❹ using the `REF` to reference the filename in the variable and then quit Word ❺.

DISCUSSION

This approach could be used to document machine information. The information is produced by utilizing the WMI scripts described in the first part of the document. It's written into a Word document by a variation of this script.

TECHNIQUE 60 Creating a configuration report

This is the fun part of the chapter where we combine everything we've learned previously to produce a Word document from a PowerShell script that will document our system. The document will be produced using the COM approach we've used so far, and as a bonus we'll look at using the OpenXML format for Word documents that was introduced with Office 2007.

The OpenXML document format is in effect a set of XML files within a zip file. The files can be examined by changing the extension (DOCX) of a Word 2007 document to ZIP. Opening the zip file allows the individual files to be read. We need to use XPath functionality to work with these files. XPath is a technology not usually associated with administrators! A tutorial on XPath can be found at http://www.w3schools.com/XPath/default.asp.

Before we can work with OpenXML documents, we need to load the OpenXML Power Tools. The binaries can be downloaded from the links in appendix E. First off, we'll work with the Word COM object.

PROBLEM

A configuration report must be produced for a system. The report must be created in a Word document for distribution.

SOLUTION

The configuration information can be obtained by using the WMI scripts from the first section of the chapter. This script brings together a number concepts starting with a function ❶. The function takes three parameters that contain the font name, the font size, and the text to be written into the document, as shown in listing 7.22. Within the body of the function, the font name and size are set. The text is written and the paragraph is closed.

Listing 7.22 Create a configuration report

```
function add-data {                    <-- ❶ Write data
param ($font, $size, $text)
$word.Selection.Font.Name = $font
$word.Selection.Font.Size = $size
$word.Selection.TypeText("$text")
$word.Selection.TypeParagraph()
}
```

```
$hdfont = "Cambria"
$hdSize = "13"
$txfont = "Courier"
$txSize = "8"

$word = New-Object -ComObject "Word.application"
$word.visible = $true
$doc = $word.Documents.Add()
$doc.Activate()

$name = (Get-WmiObject -Class Win32_ComputerSystem).Name
add-data "Cambria" "14" "Configuration report for $name"

$text = "Computer System"
add-data $hdfont $hdsize $text
$text = Get-WmiObject -Class Win32_ComputerSystem |
Select Name, Manufacturer, Model, CurrentTimeZone,
TotalPhysicalmemory | Out-String
add-data $txfont $txsize $text.Trim()

$text = "Operating System"
add-data $hdfont $hdsize $text
$text = Get-WmiObject -Class Win32_OperatingSystem |
Select Name, Version,ServicePackMajorVersion,
ServicePackMinorVersion, Manufacturer, WindowsDirectory,
Locale, FreePhysicalMemory, TotalVirtualMemorySize,
FreeVirtualMemory | Out-String
add-data $txfont $txsize $text.Trim()

$text = "Processor"
add-data $hdfont $hdsize $text
$text = Get-WmiObject -Class Win32_Processor |
select Manufacturer, Name, NumberOfCores,
NumberofLogicalProcessors, Version,
 L2CacheSize, DataWidth | Out-String
add-data $txfont $txsize $text.Trim()

$text = "BIOS"
add-data $hdfont $hdsize $text
$text = Get-WmiObject -Class Win32_Bios |
Select -Property BuildNumber, CurrentLanguage,
InstallableLanguages, Manufacturer, Name, PrimaryBIOS,
ReleaseDate, SerialNumber, SMBIOSBIOSVersion,
SMBIOSMajorVersion, SMBIOSMinorVersion, SMBIOSPresent,
Status, Version | Out-String
add-data $txfont $txsize $text.Trim()

$text = "Page File"
add-data $hdfont $hdsize $text
$text = Get-WmiObject -Class Win32_PageFileUsage |
Select AllocatedBaseSize, CurrentUsage, Description,
InstallDate, Name, PeakUsage | Out-String
add-data $txfont $txsize $text.Trim()

$text = "IP Address"
add-data $hdfont $hdsize $text
$text = Get-WmiObject -Class Win32_NetWorkAdapterConfiguration `
-Filter "IPEnabled = True" |
```

2 Set constants

3 Create document

4 Write header

5 Write configuration data

```
Select DNSHostName, Caption, MACaddress, IPAddress,
IPSubNet, DefaultIPGateway, DHCPEnabled, DHCPServer,
DHCPLeaseObtained, DHCPLeaseExpires, DNSServerSearchOrder,
DNSDomainSuffixSearchOrder, WINSPrimaryServer,
 WINSSecondaryServer | Out-String
add-data $txfont $txsize $text.Trim()

$file = "c:\scripts\office\$name config report .doc"     <-- 6 Save and close
$doc.SaveAs([REF]$file)
$Word.Quit()
```

The body of the script starts by defining some constants ❷ for font names and sizes. Note that the sizes are defined as strings rather than integers. A Word document is created ❸ using the technique from the previous script.

At this point, we can start adding our configuration information into the document. The computer name is retrieved using the `Win32_ComputerSystem` class ❹ and substituted into a string that's passed into the function we defined at the beginning of the script.

FUNCTION DEFINITIONS Functions must be defined before they're used. Best practice is to do it at the beginning of the script.

The configuration data can now be added to the document. I've chosen to use the WMI scripts from earlier in this chapter. Other WMI classes are available. The script is modular, so it's easy add or change the WMI classes used to derive the configuration information. For each set of configurations ❺, we define a title such as `"Computer System"` that we write using the header font and size we defined in the constants ❷. Using `Get-WmiObject` with our chosen class, we select the properties we want to record. The information is piped into `Out-String` so that a string is available to write into the Word document, rather than trying to write the object! The `add-data` function is used to write the data as previously, but we use the `Trim` method to remove the whitespace from before and after the data. This isn't essential, but it does make the report look better.

The Courier font is used for the data to ensure that spacing and formatting is preserved. Courier is a fixed-space font, so each character takes up the same amount of space. If a proportional spaced font such as Calibri (default in Word 2007) is used, the formatting will be lost.

When all of the configuration data has been written into the file, we need to create a filename that incorporates the machine name ❻ and then save and close the file. The configuration reports can be kept in a common folder. As WMI can be used to access remote machines, the computer name can be used in the `Get-WmiObject` cmdlet. The computer name can be passed into the script as a parameter.

DISCUSSION

As an alternative to using the COM object, we can create a Word document using the OpenXML format, as in listing 7.23. This may seem more complicated than using the more traditional COM object (and it is), but by adopting a standard approach using functions, we can make it straightforward. By comparing the two methods in this way, we can use a method we understand to help us understand the new method.

Listing 7.23 Using OpenXML

```
function add-header{
param ($text)
$content = "<w:p><w:r><w:t>$text</w:t></w:r></w:p>"
Add-OpenXmlContent -Path $file -PartPath '/word/document.xml' `
-InsertionPoint '/w:document/w:body/w:p[last()]|/w:styles/w:style[1]' `
-Content $content -SuppressBackups

Get-OpenXmlStyle -Path $template | `
Set-OpenXmlContentStyle -Path $file `
-InsertionPoint  '/w:document/w:body/w:p[last()]' `
-StyleName 'Heading2' -SuppressBackups
}

function add-text{
Get-Content data.txt | foreach {
if ($_ -ne "") {
   $content = "<w:p><w:r><w:t>$_</w:t></w:r></w:p>"
   Add-OpenXmlContent -Path $file -PartPath '/word/document.xml' `
   -InsertionPoint '/w:document/w:body/w:p[last()]|/w:styles/w:style[1]' `
   -Content $content -SuppressBackups

  Get-OpenXmlStyle -Path $template | `
  Set-OpenXmlContentStyle -Path $file `
-InsertionPoint  '/w:document/w:body/w:p[last()]' `
  -StyleName 'ConfigData' -SuppressBackups
 }
}
}

$date = (Get-Date).ToString()
$name = (Get-WmiObject -Class Win32_ComputerSystem).Name
$file = "c:\scripts\office\$name config report.docx"
Export-OpenXmlWordprocessing
-Text "Configuration report for  $name   at   $date"
-OutputPath $file

$template = "C:\Scripts\PowerShellinPractice\Chapter 07\Template.docx"
Get-OpenXmlStyle -Path $template | Set-OpenXmlStyle -Path $file
-SuppressBackups

Get-OpenXmlStyle -Path $template | `
Set-OpenXmlContentStyle -Path $file -InsertionPoint  '/w:document/w:body/
     w:p[1]' `
-StyleName 'Heading1' -SuppressBackups

add-header "Computer System"
Get-WmiObject -Class Win32_ComputerSystem |
Select Name, Manufacturer, Model,
CurrentTimeZone, TotalPhysicalmemory |
Out-File data.txt
add-text

add-header "Operating System"
Get-WmiObject -Class Win32_OperatingSystem |
Select Name, Version,ServicePackMajorVersion,
ServicePackMinorVersion, Manufacturer,
WindowsDirectory, Locale, FreePhysicalMemory,
```

1 Add paragraph header

2 Add paragraph text

3 Create file name

4 Open template file

5 Set report header

6 Create report items

```
TotalVirtualMemorySize, FreeVirtualMemory |
Out-File data.txt
add-text

add-header "Processor"
Get-WmiObject -Class Win32_Processor |
select Manufacturer, Name, NumberOfCores,
NumberofLogicalProcessors, Version, L2CacheSize,
DataWidth | Out-File data.txt
add-text

add-header "BIOS"
Get-WmiObject -Class Win32_Bios |
Select -Property BuildNumber, CurrentLanguage,
InstallableLanguages, Manufacturer, Name,
PrimaryBIOS, ReleaseDate, SerialNumber,
SMBIOSBIOSVersion, SMBIOSMajorVersion,
SMBIOSMinorVersion, SMBIOSPresent, Status,
 Version | Out-File data.txt
add-text

add-header "Page File"
Get-WmiObject -Class Win32_PageFileUsage |
Select AllocatedBaseSize, CurrentUsage, Description,
InstallDate, Name, PeakUsage | Out-File data.txt
add-text

add-header "IP Address"
Get-WmiObject
-Class Win32_NetWorkAdapterConfiguration
-Filter "IPEnabled = True" |
Select DNSHostName, Caption, MACaddress, IPAddress,
IPSubNet, DefaultIPGateway, DHCPEnabled, DHCPServer,
DHCPLeaseObtained, DHCPLeaseExpires, DNSServerSearchOrder,
DNSDomainSuffixSearchOrder, WINSPrimaryServer,
WINSSecondaryServer | Out-File data.txt
add-text
```

❻ **Create report items**

The script starts by defining two functions. One function ❶ is used to write the paragraph headers in the report; the second ❷ is used to add the body of the text in the paragraph. The functions will be explained later in the context of creating the report.

The body of the script opens by using `Get-Date` ❸ to retrieve the current date. A filename is constructed after we've retrieved the computer name. The `Export-OpenXMLWordProcessing` cmdlet is used to create the document. We need to give the cmdlet some text to put into the document and the filename.

When using the COM model in the previous script, we set the style of the text, the font and size, before we wrote the text. In the OpenXML format, we set the style after the text has been written. We have to define a template file ❹ from which we can use the styles defined in it.

TEMPLATE FILE The template file I used in this example is available in the scripts download for this chapter.

We get the styles from the template file and put them into the report file we're creating. Note the use of the `-SuppressBackups` parameter ❺. This prevents a backup of the file

being generated. I recommend using this parameter to stop your disk filling with lots of copies of your file. The default setting is that a backup is produced at every change.

Having defined the template document, we can use `Get-OpenXmlStyle` and `Set-OpenXmlContentStyle` to set the font and size of the text. The `-InsertionPoint` parameter takes an XPath definition, which reads as the first paragraph in the body of the document.

After this point, the report creation consists of defining the paragraph header ❻ and passing it to the `add-header` function, followed by the use of a WMI class to generate the report text. In this case, we have to write the text to a file and then call the `add-text` function.

`add-header` takes the text as a parameter and puts it into the correct XML tags to define its position in the paragraph. `Add-OpenXmlContent` writes the XML into the file. Remember that a DOCX file is a compressed set of XML files. The actual text is stored in the document.xml file in the Word folder. An insertion point of the last paragraph is used to ensure that the data doesn't get overwritten. Backups are suppressed. The second part of the function sets the style of this text as discussed earlier.

The `add-text` function is similar, except it reads the contents of the file and for each record checks whether the string is empty. If it's not empty, the string is written into the document and the font set to Courier by using the `ConfigData` style.

In some ways, the OpenXML format is easier to use, and in others it's harder. It's the format that will be most used in the future. Wise administrators will ensure that they have at least an understanding of how to work with it.

7.5 Summary

Administering the desktop estate in our environments can consume a large proportion of our time and effort. Much of the effort is spent discovering basic information about the systems showing problems. We can automate the discovery process using PowerShell and WMI. This can consist of a standard script returning a basic set of information and scripts that enable us to dig deeper to resolve particular issues. The issue can then be resolved using WMI and PowerShell, providing us with a powerful discovery and resolution method.

The user's desktop can be investigated using COM functionality to access the special folders. We can discover how things are configured, and then manipulate and change them as required.

The Office applications are extremely widespread in the Windows environment. We can create and access documents using these applications in PowerShell. This enables us to produce a reporting and documentation system for our machines based on using PowerShell with WMI and COM.

There are other aspects of a machine configuration and well-being, such as processes, services, event logs, and the registry that need to be included in this framework. These will be covered in the next chapter, which has sections that could also be applied to the desktop. Likewise, a large part of the material in this chapter could be applied to servers as well as desktops.

WINDOWS ADMINISTRATION

PowerShell IN PRACTICE

Richard Siddaway

PowerShell is a powerful scripting language that lets you automate Windows processes you now manage by hand. It will make you a better administrator.

PowerShell in Practice covers 205 individually tested and ready-to-use techniques, each explained in an easy problem/solution/discussion format. The book has three parts. The first is a quick overview of PowerShell. The second, Working with People, addresses user accounts, mailboxes, and desktop configuration. The third, Working with Servers, covers techniques for DNS, Active Directory, Exchange, IIS, and much more. Along the way, you'll pick up a wealth of ideas from the book's examples: 1-line scripts to full-blown Windows programs.

What's Inside

- Basics of PowerShell for sysadmins
- Remotely configuring desktops and Office apps
- 205 practical techniques

This book requires no prior experience with PowerShell.

Richard Siddaway is an IT Architect with 20 years of experience as a server administrator, support engineer, DBA, and architect. He is a PowerShell MVP and founder of the UK PowerShell User Group

For online access to the author, and a free ebook for owners of this book, go to www.manning.com/PowerShellinPractice

"A definitive source."
—Wolfgang Blass
Microsoft Germany

"A must read!"
—Peter Johnson
Unisys Corp.

"A new perspective on PowerShell!"
—Andrew Tearle
Thoughtware N.Z.

"Real-world examples... in a language you can understand."
—Marco Shaw
Microsoft MVP

ISBN 13: 978-1-935182-00-9
ISBN 10: 1-935182-00-5
54999
9 781935 182009

MANNING $49.99 / Can $62.99 [INCLUDING eBOOK]