Chapter 8

Automation Using PowerShell

Virtual Machine Manager is one of the first Microsoft software products to fully adopt Windows PowerShell and offer its users a complete management interface tailored for scripting. From the first release of VMM 2007, the Virtual Machine Manager Administrator Console was written on top of Windows PowerShell, utilizing the many cmdlets that VMM offers. This approach made VMM very extensible and partner friendly and allows customers to accomplish anything that VMM offers in the Administrator Console via scripts and automation. Windows PowerShell is also the only public application programming interface (API) that VMM offers, giving both developers and administrators a single point of reference for managing VMM. Writing scripts that interface with VMM, Hyper-V, or Virtual Server can be made very easy using Windows PowerShell's support for WMI, .NET, and COM.

In this chapter, you will learn to:

- Describe the main benefits that PowerShell offers for VMM
- Use the VMM PowerShell cmdlets
- Create scheduled PowerShell scripts

VMM and Windows PowerShell

System Center Virtual Machine Manager (VMM) 2007, the first release of VMM, was one of the first products to develop its entire graphical user interface (the VMM Administrator Console) on top of Windows PowerShell (previously known as Monad). This approach proved very advantageous for customers that wanted all of the VMM functionality to be available through some form of an API. The VMM team made early bets on Windows PowerShell as its public management interface, and they have not been disappointed with the results. With its consistent grammar, the great integration with .NET, and the abundance of cmdlets, PowerShell is quickly becoming the management interface of choice for enterprise applications.

Unlike other traditional public APIs that focus on developers, VMM's PowerShell interface is designed for the administrator. With the extensive help contents and the well-documented System Center Virtual Machine Manager Scripting Guide that is available from the Microsoft Download Center, the VMM team positioned its cmdlets to be the premier way of scripting in your virtualization environment. In addition to the VMM cmdlets, your scripts can be enhanced by the built-in support that Windows PowerShell has for a variety of data stores, like the filesystem, the Registry, and WMI.

Installing the VMM PowerShell Cmdlets

Even though the VMM PowerShell interface does not have its own installer, it is always installed as part of the VMM Administrator Console setup. VMM Setup will install the 32-bit version of the Administrator Console on 32-bit systems and the 64-bit version of the Administrator Console on 64-bit systems. Due to the nature of the VMM cmdlets, some utilize both native and .NET binaries in their implementation. This approach prohibits the VMM PowerShell cmdlets from being architecture independent, which means that only 32-bit PowerShell cmdlets will work on a 32-bit system. The 64-bit PowerShell cmdlets have the same issue and will work on only a 64-bit system. Any process that attempts to load the PowerShell runspace and invoke the VMM cmdlets needs to be aware of this restriction and factor this limitation in the design.

VMM 2008 is certified to work with both Windows PowerShell version 1.0 and version 2.0. Windows PowerShell is already included as part of Windows Server 2008 and Windows Server 2008 R2, and it can be downloaded for free from the Microsoft website at the following location:

http://www.microsoft.com/windowsserver2003/technologies/management/powershell/download.mspx

Windows PowerShell 1.0 officially supports Windows XP Service Pack 2, Windows Server 2003, Windows Vista, and Windows Server 2008. Windows PowerShell 2.0 was released with the Windows 7 and Windows Server 2008 R2 operating systems. PowerShell 2.0 will be eventually pack-ported to other operating systems as well.

When you first launch Windows PowerShell in your system and import the VMM PowerShell snap-in, you will be prompted to add Microsoft Corporation to the list of trusted publishers, as per Figure 8.1. Enter **A** in this case for Always Run.

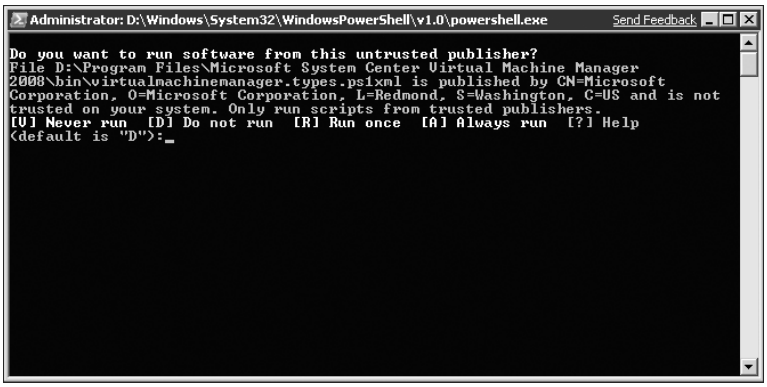


FIGURE 8.1 Adding Microsoft Corporation to the list of trusted publishers for Windows PowerShell scripts

It is possible Windows PowerShell will still prevent you from running scripts in your system, so you need to set the proper execution policy for scripts before any Windows PowerShell scripts are allowed to run on your computer. This can be achieved using the `Set-ExecutionPolicy` cmdlet. Figure 8.2 shows the help contents of this cmdlet. Type in **`get-help Set-ExecutionPolicy -detailed | more`** for more information on the policy options. To invoke the `Set-ExecutionPolicy` cmdlet, you need to run Windows PowerShell as Administrator.

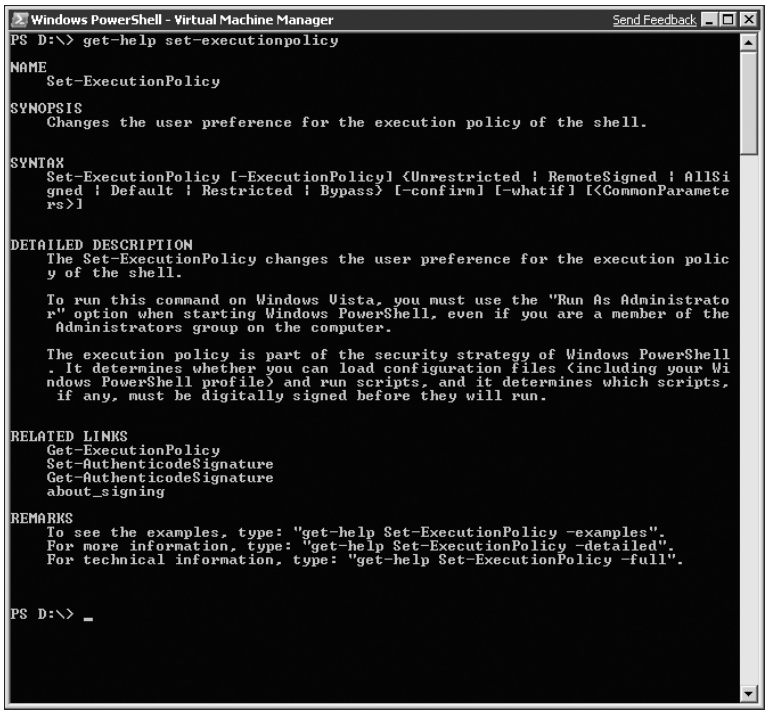


FIGURE 8.2
Setting the execution policy for Windows PowerShell

Exposing the VMM Cmdlets

To get started with VMM and Windows PowerShell, open a console window that has the VMM PowerShell snap-in loaded. There are a few ways to accomplish this, as shown here:

1. Click Start ➤ All Programs ➤ Microsoft System Center ➤ Virtual Machine Manager 2008 R2 and launch Windows PowerShell - Virtual Machine Manager. This command will automatically launch Windows PowerShell version 1.0 and pass as input the VMM PowerShell Console file located at

   ```
   %SystemDrive%\Program Files\Microsoft System Center Virtual
    Machine Manager 2008 R2\bin\cli.psc1
   ```

2. Open a regular Windows PowerShell console window and add the VMM snap-in using the following command. Once the VMM snap-in is added, you can use all the VMM cmdlets.

   ```
   Add-PSSnapin "Microsoft.SystemCenter.VirtualMachineManager"
   ```

3. Windows PowerShell can also be launched from the Administrator Console by clicking the PowerShell button in the toolbar of the main view.

Once the VMM PowerShell snap-in is added, you can get a list of all VMM cmdlets using the `get-command` cmdlet.

```
# This cmdlet will get all cmdlets that can be executed by the currently
# loaded PowerShell snapins
Get-command

# This cmdlet will list only the cmdlets exposed by the VMM Windows
# PowerShell snapin
get-command -module "Microsoft.SystemCenter.VirtualMachineManager"
```

Getting Help on VMM Cmdlets

If you have the name of a VMM cmdlet, you can get more information on it, including a list of examples. The following code shows how to get the definition of the `Refresh-VM` cmdlet:

```
PS D:\> get-command refresh-vm

CommandType     Name                            Definition
-----------     ----                            ----------
Cmdlet          Refresh-VM                      Refresh-VM [-VM] <VM> [-RunA...
```

The following code shows how to get the detailed view of the parameters that can be passed to the `Refresh-VM` cmdlet:

```
PS D:\> get-command refresh-vm | format-list

Name            : Refresh-VM
CommandType     : Cmdlet
Definition      : Refresh-VM [-VM] <VM> [-RunAsynchronously] [-JobVariable <St
                  ring>] [-PROTipID <Nullable`1>] [-Verbose] [-Debug] [-ErrorA
                  ction <ActionPreference>] [-WarningAction <ActionPreference>
                  ] [-ErrorVariable <String>] [-WarningVariable <String>] [-Ou
                  tVariable <String>] [-OutBuffer <Int32>]

Path            :
AssemblyInfo    :
DLL             : D:\Program Files\Microsoft System Center Virtual Machine Man
                  ager 2008 R2\bin\Microsoft.SystemCenter.VirtualMachineManager.d
                  ll
HelpFile        : Microsoft.SystemCenter.VirtualMachineManager.dll-Help.xml
ParameterSets   : {[-VM] <VM> [-RunAsynchronously] [-JobVariable <String>] [-P
                  ROTipID <Nullable`1>] [-Verbose] [-Debug] [-ErrorAction <Act
                  ionPreference>] [-WarningAction <ActionPreference>] [-ErrorV
                  ariable <String>] [-WarningVariable <String>] [-OutVariable
                  <String>] [-OutBuffer <Int32>]}
ImplementingType : Microsoft.SystemCenter.VirtualMachineManager.Cmdlets.Refresh
                  VmCmdlet
```

```
Verb             : Refresh
Noun             : VM
```

The following code shows how to get the help for the `Refresh-VM` cmdlet:

```
PS D:\> get-help refresh-vm

NAME
    Refresh-VM

SYNOPSIS
    Refreshes the properties of a virtual machine so that the Virtual Machine M
    anager Administrator Console displays updated information about the virtual
     machine.

SYNTAX
    Refresh-VM [-VM] [<String VM>] [-JobVariable <String>] [-PROTipID <Guid>] [
    -RunAsynchronously] [<CommonParameters>]

DETAILED DESCRIPTION
    Refreshes the properties of a virtual machine so that the Virtual Machine M
    anager Administrator Console displays updated information about the virtual
     machine. The updated properties include Name, Location, Status, OperatingS
    ystem, and other properties.

RELATED LINKS
    Get-VM
    Refresh-LibraryShare
    Refresh-VMHost

REMARKS
    To see the examples, type: "get-help Refresh-VM -examples".
    For more information, type: "get-help Refresh-VM -detailed".
    For technical information, type: "get-help Refresh-VM -full".
```

The next code shows the different parameters that can be passed to the `get-help` cmdlet:

```
# To show detailed information on the refresh-vm cmdlet, use the
# -detailed parameter
PS D:\> get-help Refresh-VM -detailed

# This command will show examples on using the cmdlet refresh-vm
PS D:\> get-help Refresh-VM -examples

# This command will show the full information on using the refresh-vm cmdlet
PS D:\> get-help Refresh-VM -full
```

Some VMM cmdlets offer different parameters sets. In Figure 8.3 and Figure 8.4, you can see the different parameter sets for the `Get-VM` cmdlet. For example, you can get a

list of virtual machines (VMs) by a matching name, by using the ID of the VM object, or by executing the cmdlet against a specific host. Any parameter that is included in square brackets is optional. If a required parameter is not specified, PowerShell will prompt for it as Figure 8.5 shows.

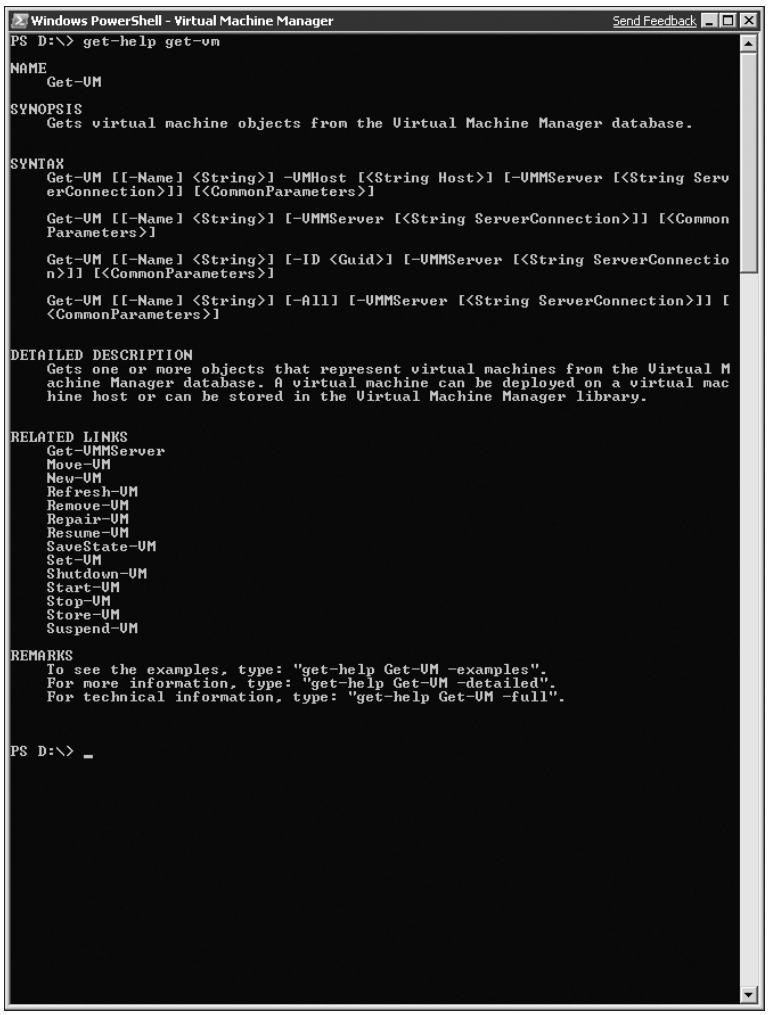


FIGURE 8.3
Get-VM's different parameter sets

Using the VMM Cmdlets

By now you know how to get a list of all VMM cmdlets and their descriptions. To get more information about each cmdlet, including details on each parameter and examples on how to invoke them, you can use the `get-help` cmdlet with the `-full`, `-examples`, or `-detailed` parameter.

FIGURE 8.4
Get-VM's required parameters

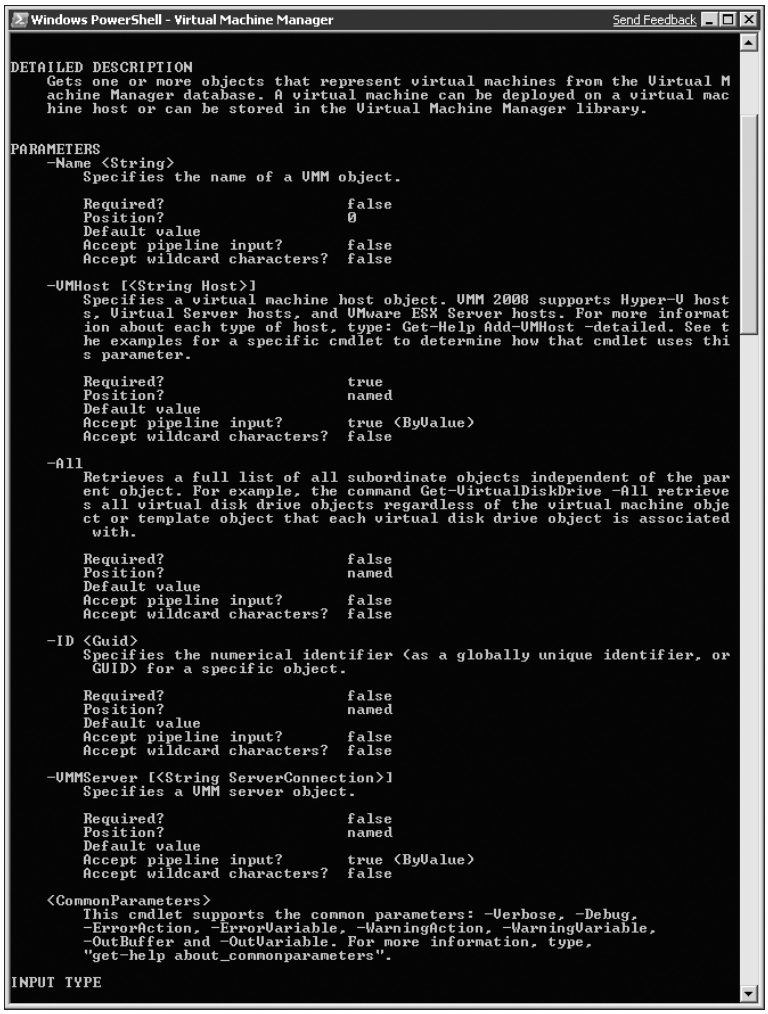


FIGURE 8.5
Windows PowerShell will prompt for required parameters.

The Virtual Machine Manager Scripting Guide is also a great reference for learning how to use Windows PowerShell with VMM. VMM published a scripting guide with both VMM 2007 and VMM 2008. You can find the scripting guides at `http://www.microsoft.com/downloads/details.aspx?familyid=3DA5BA7E-AD72-4D2C-B573-1B74894D1DDF&displaylang=en` and at `http://technet.microsoft.com/en-us/library/cc764259.aspx`. A scripting guide update for VMM 2008 R2 should be available on the System Center Virtual Machine Manager TechCenter shortly.

Appendix B, ''VMM Windows PowerShell Object Properties and VMM Cmdlet Descriptions,'' contains a reference list of Virtual Machine Manager cmdlets and a short description of their functionality.

Noun Properties

Windows PowerShell uses a verb-noun format for the names of the cmdlets. The verb identifies the action to be performed, such as *get*, *add*, *set*, *remove*, or *new*. The Windows PowerShell team has a published list of common verbs for cmdlet developers to adhere to.

The noun identifies the type of object on which the cmdlet will operate. Example nouns include `VM` (virtual machine), `VMHost` (virtualization host), `VMHostCluster` (Failover Cluster), and `VirtualNetwork` (virtual network). In this section, we will look at three of the most frequently used nouns in VMM and explain their property values.

VMM cmdlets will in most cases return back to the pipeline the noun of the cmdlet. This allows you to use the pipeline and combine multiple VMM cmdlets for more complicated scripts. For `Get-*` cmdlets, like `Get-VM` for example, it is possible for the value that is returned to the pipeline to be a collection instead of a single object. You can check the type of the return value using the code snippet in the next paragraph.

POWERSHELL PIPELINE

The PowerShell pipeline is similar in concept to the ''pipeline'' seen in Unix shell scripting environments. The pipeline allows you to create a multitude of single-purpose and easy-to-understand cmdlets and then combine them to achieve a bigger task (just like building blocks).

The following code shows how to invoke the `Get-VM` cmdlet and check the result:

```
# Execute Get-VM on a VM that does not exist. In this case $vm should be null
$vm = Get-VM "VMDoesNotExist"
$vm -eq $null

# Execute Get-VM as a targeted get for a single VM. In this case
# the result should not be an array
$vm = Get-VM "virtualmachine1"
$vm -is [Array]

# Execute Get-VM to get all VMs in the system. In this case the
# result may be an array
$vm = Get-VM
$vm -is [Array]
```

VMMServer Object

VMMServer represents the object that contains the connection to the Virtual Machine Manager. This object also contains some of the global settings of the VMM server installation and environment. Once a connection to the VMM server is established, the connection is cached and future cmdlets that need a connection object will automatically use the existing cached connection.

To see the `VMMServer` noun and a list of its properties with an explanation for each property, see Appendix B.

VM Object

`VM` represents the object that contains a virtual machine instance in Virtual Machine Manager.

To see the virtual machine noun and its properties, see Appendix B. Each property also contains a definition for its value or values.

VMHost Object

`VMHost` represents the object that contains a physical computer that is a virtual machine host. A virtual machine host could be a Virtual Server host, a Hyper-V host, or a VMware ESX host. The following code shows you how to get a list of all the properties of a host and inspect their values:

```
PS D:\> $vmhost = Get-VMHost "hostname"
PS D:\> $vmhost

Windows PowerShell will output all the properties and values for the
 supplied host.
```

To see the `VMHost` noun and its properties, see Appendix B. Each property also contains a definition for its value or values.

Leveraging the Public PowerShell API

Virtual Machine Manager uses the Windows PowerShell cmdlets for VMM as the single point of entry for VMM. Developers can integrate with VMM programmatically by leveraging the publicly available VMM PowerShell cmdlets. Calling cmdlets programmatically is not much different than invoking them in a Windows PowerShell command window. In this section, we will look at programmatically invoking the cmdlets and give you example code to achieve this type of integration with VMM.

Programmatically Calling the VMM Cmdlets

The set of Windows PowerShell cmdlets for VMM is the only supported API for integrating with VMM. To programmatically call the cmdlets and manage VMM, you would need to create a PowerShell runspace and invoke the cmdlets in the same way you would if you were using the Windows PowerShell command window.

To start, you need to know the path to the VMM assemblies that are needed to resolve the VMM objects and cmdlets. You can find the installation path for the Virtual Machine Manager Administrator Console under the Registry key `HKLM\Software\Microsoft\Microsoft System Center Virtual Machine Manager Administrator Console\Setup`. The `InstallPath` value of

this Registry key contains the root of the installation path for VMM. The four assemblies that are needed for the programmatic usage of VMM cmdlets are located in the `bin` folder of the installation directory. These binaries are listed here:

- `Microsoft.SystemCenter.VirtualMachineManager.dll`
- `Remoting.dll`
- `Utils.dll`
- `Errors.dll`

Because none of these VMM assemblies are listed in the Global Assembly Cache (GAC), if you want your application to be able to resolve them without copying the binaries, you need to use an assembly resolve handler. The following code shows you how to add such a handler and how to properly load the correct assembly. The `InstallPath` value from the Registry is needed for this purpose:

```
// Add the code for the assembly resolver before any of the VMM binaries
are invoked.
Thread.GetDomain().AssemblyResolve += new
 ResolveEventHandler(VMMResolveEventHandler);

// In this example, we automatically resolve everything to
the VirtualMachineManager DLL. A smart resolver would look
at the args.Name and match the assembly to be resolved with
its proper full path.
static Assembly VMMResolveEventHandler(object sender, ResolveEventArgs args)
{
   return Assembly.LoadFrom("D:\Program Files\Microsoft System
 Center Virtual Machine Manager 2008 R2\bin\
 Microsoft.SystemCenter.VirtualMachineManager.dll");
}
```

In addition to the required binaries, the WCF port number of the VMM server is needed to establish a successful connection. You can find the TCP port number for WCF under the VMM server Registry key `HKLM\Software\Microsoft\Microsoft System Center Virtual Machine Manager Server\Settings`. The `IndigoTcpPort` value contains the port number that all VMM clients need to use to connect.

Now we have all the data we need to connect to VMM and execute some PowerShell cmdlets. It is recommended to use a utility to create a wrapper on top of the VMM PowerShell snap-in. Such a wrapper would produce a familiar .NET interface for developers and provide type safety for all PowerShell cmdlets. There are a few publicly available tools to create such wrappers. In the following example, I am using the native PowerShell implementation, directly invoking cmdlets using their string names. You can see that any mistakes in this code will not be caught by the compiler.

Listing 8.1 is a complete program that creates a PowerShell runspace, adds the VMM Power-Shell snap-in to it, creates a connection to the VMM server on default port 8100, and gets a list of all VMs that the current user is authorized to see. The list of VMs returned is written on the console window as in Figure 8.6.

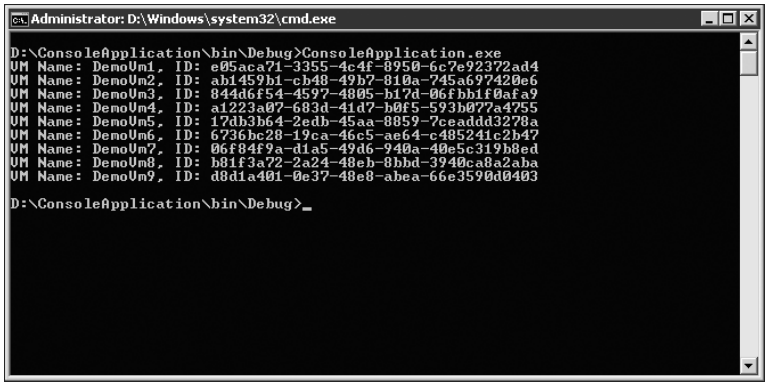


FIGURE 8.6
Output from the programmatic invocation of Get-VM

LISTING 8.1: Invoking the VMM cmdlets programmatically

```
// Add the proper namespaces
using System;
using System.Management.Automation;
using Microsoft.VirtualManager.Utils;
using Microsoft.SystemCenter.VirtualMachineManager;
using Microsoft.SystemCenter.VirtualMachineManager.Cmdlets;
using Microsoft.SystemCenter.VirtualMachineManager.Remoting;
using Microsoft.VirtualManager.Remoting;
using System.Management.Automation.Runspaces;
using System.Collections.ObjectModel;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a default RunspaceConfiguration
            RunspaceConfiguration config = RunspaceConfiguration.Create();

            // Add the VMM PowerShell snapin
            PSSnapInException warning = null;
            config.AddPSSnapIn("Microsoft.SystemCenter.VirtualMachineManager",
out warning);
            if (warning != null)
            {
                Console.WriteLine(warning.Message);
                return;
            }
```

```
                // Create the Runspace using this configuration
                Runspace runspace = RunspaceFactory.CreateRunspace(config);
                try
                {
                    runspace.Open();
                    Command psCommand = null;
                    ServerConnection serverConnection = null;
                    VM vmObject = null;

                    // Create a Pipeline
                    using (Pipeline pipeline = runspace.CreatePipeline())
                    {
                        // Call the get-vmmserver cmdlet to get a connection to VMM
                        // If you plan to use VMM as a platform and develop
                        // a separate GUI on top of VMM, you might also want to
                        // set the RetainDeletedObjects and RetainObjectCache
                        // properties of the Get-VMMServer cmdlet. For more
                        // information on these properties, look at the help
                        // for the cmdlet.
                        psCommand = new Command("Get-VMMServer");
                        psCommand.Parameters.Add("ComputerName", "localhost");
                        psCommand.Parameters.Add("TCPPort", "8100");
                        pipeline.Commands.Add(psCommand);

                        // Invoke the cmdlet
                        Collection<PSObject> psObjList = pipeline.Invoke();
                        serverConnection =
(ServerConnection)(psObjList[0].BaseObject);
                    }

                    // Create a Pipeline
                    using (Pipeline pipeline = runspace.CreatePipeline())
                    {
                        // Call the get-vm cmdlet to get all VMs in the system
                        psCommand = new Command("Get-VM");
                        pipeline.Commands.Add(psCommand);

                        // Invoke the cmdlet
                        Collection<PSObject> psObjList = pipeline.Invoke();

                        // Enumerate the results of the cmdlet
                        foreach (PSObject obj in psObjList)
                        {
                            vmObject = (VM)obj.BaseObject;
                            Console.WriteLine("VM Name: {0},
ID: {1}", vmObject.Name, vmObject.ID);
```

```
                    }
                }
            }
            finally
            {
                // Close the runspace if it is already open
                if (runspace.RunspaceStateInfo.State == RunspaceState.Opened)
                {
                    runspace.Close();
                }
            }
        }
    }
}
```

When trying to programmatically code against VMM, it is useful to understand and know all the different values and properties for the various VMM classes and enumerations. To get familiar with the nouns and their properties, you can load the `Microsoft.SystemCenter.VirtualMachineManager.dll` into the Object Browser of Visual Studio and start looking into the classes defined in this assembly.

The following code snippet shows the `Status` property of a virtual machine object as defined by the VM class. Figure 8.7 shows the list of possible values for the VMComputerSystemState enumeration.

```
public class VM
{
   public VMComputerSystemState Status { get; }
}
```

The Host class has multiple state objects. In the following code snippet, along with Figure 8.8 and Figure 8.9, we show you how to get the CommunicationState and the ComputerState of a host object and the possible values for the two enumerations.

```
public class Host
{
   public CommunicationState CommunicationState { get; }
   public ComputerState ComputerState { get; }
}
```

Creating PowerShell Scripts

PowerShell scripts are simple text files that you can author in Notepad and save with the filename extension `.ps1`. You can also choose to download a PowerShell interactive development environment (IDE). Such an IDE will accelerate development through rich syntax coloring, IntelliSense, tabbing, and rich debugging.

FIGURE 8.7
Getting the state enumeration for a VM

Microsoft.VirtualManager.Utils.VMComputerSystemState

```
using System;
using System.ComponentModel;

namespace Microsoft.VirtualManager.Utils
{
    [TypeConverter(typeof(VMStateHelper))]
    public enum VMComputerSystemState
    {
        Running = 0,
        PowerOff = 1,
        PoweringOff = 2,
        Saved = 3,
        Saving = 4,
        Restoring = 5,
        Paused = 6,
        DiscardSavedState = 10,
        Starting = 11,
        MergingDrives = 12,
        Deleting = 13,
        DiscardingDrives = 80,
        Pausing = 81,
        UnderCreation = 100,
        CreationFailed = 101,
        Stored = 102,
        UnderTemplateCreation = 103,
        TemplateCreationFailed = 104,
        CustomizationFailed = 105,
        UnderUpdate = 106,
        UpdateFailed = 107,
        UnderMigration = 200,
        MigrationFailed = 201,
        CreatingCheckpoint = 210,
        DeletingCheckpoint = 211,
        RecoveringCheckpoint = 212,
        CheckpointFailed = 213,
        InitializingCheckpointOperation = 214,
        FinishingCheckpointOperation = 215,
        Missing = 220,
        HostNotResponding = 221,
        Unsupported = 222,
        IncompleteVMConfig = 223,
        UnsupportedSharedFiles = 224,
        UnsupportedCluster = 225,
        P2VCreationFailed = 240,
        V2VCreationFailed = 250,
    }
}
```

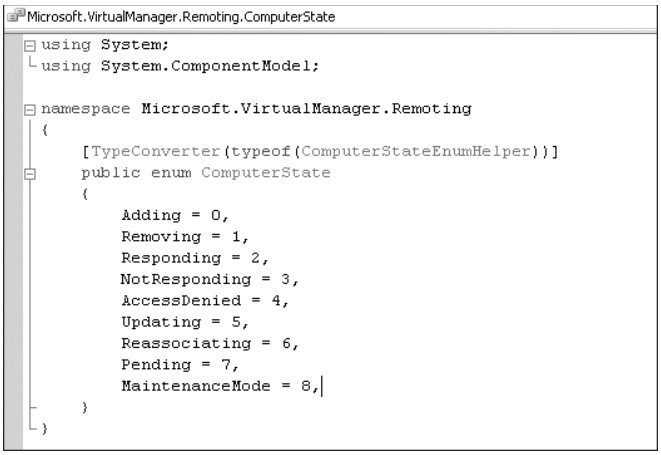
FIGURE 8.8
Getting the communication state enumeration for a Host

Microsoft.VirtualManager.Remoting.CommunicationState

```
using System;
using System.ComponentModel;

namespace Microsoft.VirtualManager.Remoting
{
    [TypeConverter(typeof(CommunicationStateEnumHelper))]
    public enum CommunicationState
    {
        Responding = 0,
        NotResponding = 1,
        AccessDenied = 2,
        Connecting = 3,
        Disconnecting = 4,
        Resetting = 5,
        NoConnection = 6,
    }
}
```

FIGURE 8.9
Getting the computer state enumeration for a Host



To create a PowerShell script file, follow these instructions:

1. Open `Notepad.exe` or your favorite editor.
2. Add the PowerShell cmdlets you want the script to execute. For example, `Get-VM | Refresh-VM`.
3. Save the file as `RefreshVirtualMachines.ps1`.

Once you have created your PowerShell script file, you can invoke it by using the full path to the file in PowerShell. If the following code snippet fails due to the PowerShell execution policy, refer to Figure 8.2 for more information:

```
PS D:\> .\demo.ps1
This is a demo script
```

Passing parameters to a PowerShell script is similar to passing them in other scripting languages. Inside the script, you can use the `$args` variable to get the parameters passed to the script:

```
PS D:\> .\demo.ps1 3 parameters passed
This is a demo script
Parameter # 1 : 3
Parameter # 2 : parameters
Parameter # 3 : passed
```

Even though creating PowerShell scripts is easy, debugging scripts is harder than debugging individual cmdlets. It is recommended to try your sequence of cmdlets in the interactive Windows PowerShell window and make sure they work before putting everything in a script file.

Windows PowerShell and the Administrator Console

In addition to the documentation on the VMM cmdlets, the Virtual Machine Manager Administrator Console provides a few other ways to get familiar with PowerShell and VMM. The additional features for enhancing your PowerShell knowledge are in the following list:

Viewing the PowerShell script that VMM will execute All VMM wizards in the Administrator Console have a View Script button in the last wizard page. Clicking it opens up Notepad and shows you the Windows PowerShell cmdlets that VMM will invoke to execute the actions the user chose in the wizard. Figure 8.10 has an example script that was generated during the migration of a VM from one host to another. This is a great way to start learning more about PowerShell and automating some of the actions that VMM provides in the wizard pages.

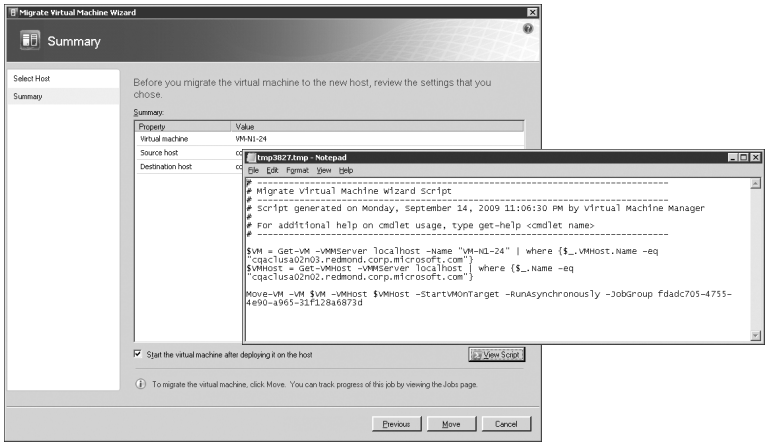


FIGURE 8.10
The View Script button at the end of the migration wizard and an example script

Viewing the PowerShell cmdlet equivalent for a VMM job The Administrator Console's jobs view has a column that lists the VMM cmdlet used for each VMM job that changes data (for example, the `Get-*` cmdlets will not have an entry in this view since a VMM job is not created for such cmdlets). Each job has a friendly name that indicates the action performed, and you can get more information on the specific cmdlet invoked through the `Get-Help` cmdlet. Figure 8.11 shows you a list of VMM jobs and their associated cmdlet names. The result type that is listed indicates the type of object on which this operation was performed. This is usually the noun of the cmdlet.

VMM library support for Windows PowerShell scripts The VMM library has built-in support for PowerShell scripts. You can place all your files with the `.ps1` filename extension in the library shares and VMM will automatically import them into the library view. From the library, you can view and edit the scripts (editing the PowerShell Scripts requires that the user of the Administrator Console has write permissions on the library share and the associated PowerShell script file), or you can execute the scripts as per Figure 8.12. Through the Run PowerShell Script action of the VMM library, VMM will launch a new Windows PowerShell process and invoke this PowerShell script. Because VMM executes your script after it obtains a connection to the VMM server, you don't need to execute `Get-VMMServer` at the beginning of your script. VMM will also keep the PowerShell window open for inspecting the results.

FIGURE 8.11
The Command column in the jobs view of the Administrator Console

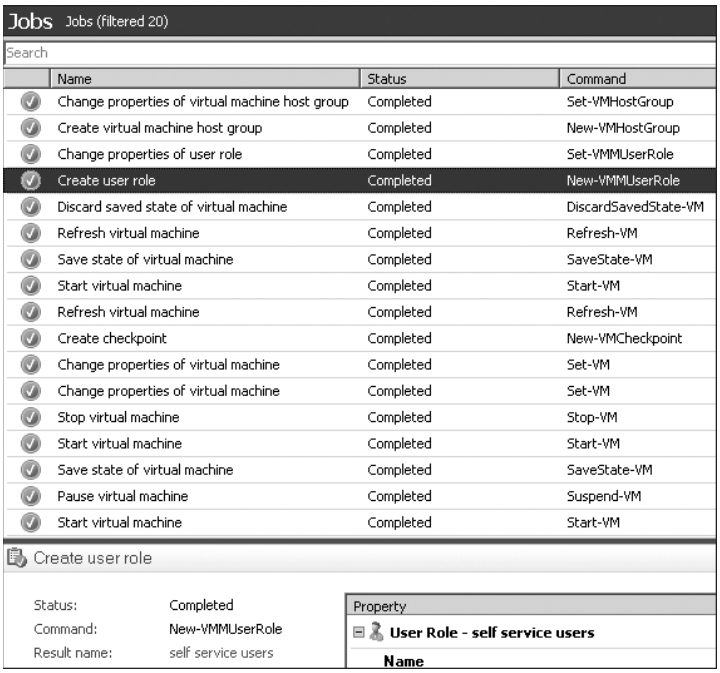
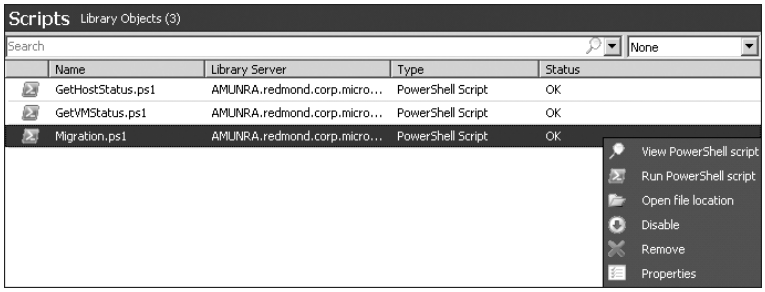


FIGURE 8.12
PowerShell scripts in the VMM library



Interfacing with Hyper-V and Virtual Server

Hyper-V has a well-documented WMI API that can be accessed from the Microsoft Developer Network's website. It is listed under the Virtualization WMI Provider documentation, and the namespace for the WMI is `root\virtualization`. The WMI provider is hosted by the Hyper-V Virtual Machine Management Windows Service. The following code shows how to get all virtual Machines from a Hyper-V server using Windows PowerShell and the WMI API for Hyper-V:

```
# Get list of Virtual Machine and host machine from Hyper-V WMI API
$computerlist = get-wmiobject Msvm_ComputerSystem -namespace root\virtualization
```

```
# Show in tabular format the list of VMs and Host
# (Host is identified in the caption as "Hosting Computer System")
# The Name for Virtual Machines contains the unique GUID that identifies a VM
# ElementName contains the user-friendly name of the VM
$computerlist | select Name, ElementName, Caption
```

A simple way to test WMI queries before executing them in Windows PowerShell is through the `wbemtest` utility. This utility is installed in the `%SystemDrive%\Windows\System32\wbem\` Windows folder. To execute a WMI query using this utility, follow these steps:

1. Launch `wbemtest.exe`.
2. Click Connect and type **`root\virtualization`** in the Namespace field.
3. Click Query and type a WMI query, such as

   ```
   select Name, ElementName, Caption from Msvm_ComputerSystem
   ```

4. The results will include the same list as the preceding PowerShell example. Click on each individual result to see the `Name`, `ElementName`, and `Caption` values.

Virtual Server's COM interface is documented on the Microsoft Developer Network under the title of Microsoft Virtual Server Reference. To be able to invoke this API using PowerShell, you need to meet the security prerequisites.

By default, PowerShell does not have the necessary COM security level to invoke the Virtual Server COM API. To accomplish that, follow these steps:

1. Create a new library DLL that allows you to set the COM security level to Impersonate for any COM object. To accomplish this, add a new API to the library DLL called `SetVSSecurity`. In this API, you need to invoke `CoSetProxyBlanket` with the `RPC_C_IMP_LEVEL_IMPERSONATE` parameter for the COM object that is passed as a parameter.
2. Once you create this DLL, you can use the `System.Reflection.Assembly.LoadFrom()` .NET API and pass the DLL's full path as a parameter.
3. Once you have a Virtual Server object in PowerShell, you need to invoke the API from this DLL to set the COM security to Impersonate so that the object can be used. The API `SetVSSecurity` should take a PowerShell object as a parameter.
4. You might need to set other objects' COM security as well, as you start working with Virtual Server and PowerShell. For example, the VM object will need to have its COM security elevated to Impersonate before it can be used. The following code shows how to get the Virtual Server root object using PowerShell.

   ```
   # Create a new Virtual Server COM instance
   $VirtualServer = new-object -com VirtualServer.Application -Strict

   # Now set the COM security of the $VirtualServer object to "Impersonate"
   # and then you can use this object to manage Virtual Server
   [Full namespace path for the DLL created in step 1 above]::SetVSSecurity
    ($VirtualServer)
   ```

```
# After setting the property of $VirtualServer to "Impersonate", you can
# get a list of Virtual Machines and use the full COM API of
# Virtual Server
```

Automating Common Tasks Using the Windows Scheduler

IT personnel today spend a lot of time on repetitive tasks to accomplish various jobs. PowerShell provides a powerful language that can be used to write and execute scripts. These scripts can eliminate repetitive tasks and add the necessary logic to complete complex jobs. Since VMM is built entirely on top of PowerShell, anything an administrator can do in the Administrator Console can also be accomplished via PowerShell cmdlets. If you combine that with the ability to integrate with .NET and other data stores that are PowerShell ready, an administrator should be able to translate a lot of manual work into PowerShell scripts. The ability to schedule PowerShell scripts at specified intervals allows an administrator to do passive management of their system and let PowerShell do some of the heavy lifting during nonworking hours.

Once you have a PowerShell script ready, you may want to execute it at regular intervals and capture its results in a log file. If the cmdlets change data in VMM, you can also view the results in the Administrator Console's jobs view. There are a couple of ways to create a scheduled task in Windows Server. In this section, we will show you how to do this from the Task Scheduler user interface. Optionally, you can use the `schtasks.exe` utility to create a scheduled task.

To schedule a task from the Task Scheduler, follow these steps:

1. Open the Task Scheduler MMC snap-in. Task Scheduler is located in either `Control Panel\System and Security\Administrative Tools\Task Scheduler` or `Control Panel\Administrative Tools\Task Scheduler`, depending on the version of Windows installed.
2. Select Create Task.
3. Enter a Task Name like **Windows PowerShell automated script**.
4. Select Run Whether User Is Logged On Or Not and chose to store the password.
5. Select Change User Or Group to enter a user that has the proper VMM privileges to execute this PowerShell script.
6. In the Triggers tab, enter the schedule you would like to create for this scheduled task. For example, you can chose to run this script daily at 8 p.m.
7. In the Actions tab, as shown in Figure 8.13, add a new action and select Start A Program. In the program path, enter **`D:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe`**.
   This is the full path to Windows PowerShell 1.0.
   For arguments, enter the following:

```
-PSConsoleFile "D:\Program Files\Microsoft System Center Virtual Machine
Manager 2008 R2\bin\cli.psc1" -Command " & '\\hypervhost1.vmmdomain.com\
MSSCVMMLibrary\Scripts\GetVMStatus.ps1'"
-NoProfile -Noninteractive
```

If you were to execute this command from a regular command window, it would look like this:

```
D:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
 -PSConsoleFile "D:\Program Files\Microsoft System Center Virtual
 Machine Manager 2008 R2\bin\cli.psc1" -Command " &
 '\\hypervhost1.vmmdomain.com\MSSCVMMLibrary\Scripts\GetVMStatus.ps1'"
 -NoProfile -Noninteractive
```

8. Click OK and enter the password for the account that will execute the scheduled task.
9. From the Task Scheduler MMC, you can view all your scheduled tasks, check for their last run time, and see if there were any errors in execution based on the last run result.

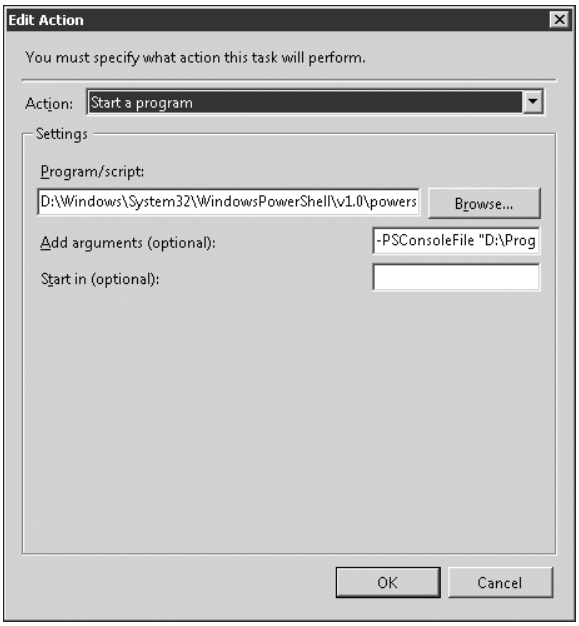


FIGURE 8.13
Adding the scheduled task action

Sometimes, it is easier to check if a scheduled task is executing by looking at a log file. The following sample PowerShell script shows you how to log that information to a file:

```
Write-Output "Script executing at " (date)

# Get a connection to the local VMM server
$c = get-vmmserver localhost

# Make a sample query to get some data from VMM
$results = get-vm | select name, status, vmid, ID, hostname
```

```
# Create a log file in the temp directory
$filepath = "$env:temp\PSscriptOutput.log"

# Append to the log file the current time and the data retrieved from VMM
Add-Content (date) -Path $filepath
Add-Content $results -Path $filepath
Add-Content "-------------" -Path $filepath
```

Because scheduled PowerShell scripts don't offer the same degree of debugging, you need to ensure that the proper execution policies are in place for PowerShell scripts. It is recommended that all scripts you execute using the Task Scheduler are signed using a code signing certificate issued by a certificate authority. This will enable you to set the PowerShell execution policy to a more secure level like the `AllSigned` option. After you sign a script using the `Set-AuthenticodeSignature` cmdlet, you will need to add the publisher of the script to your trusted publishers. PowerShell will prompt you to do that on the first execution of the script.

In Chapter 9, ''Writing a PRO Pack,'' we will cover the PRO feature of VMM. PRO allows an administrator to execute a PowerShell script or perform a VMM action based on a set of alerts detected by System Center Operations Manager (OpsMgr). OpsMgr is a comprehensive datacenter monitoring tool. In this case, a PowerShell script is executed in response to a dynamic event.

Windows PowerShell Examples

In the following sections, we will list a few different Windows PowerShell scripts that leverage the VMM cmdlets to accomplish important tasks in VMM, make it easier for an administrator to execute repetitive actions, and allow an administrator to get quick status on the health of VMM objects.

Creating Virtual Machines

There are many ways to create a virtual machine in VMM. In this section, we will take a look at an example of how to create a highly available (HA) VM and find the best suitable host on which to place it. The best suitable host is found by the Intelligent Placement feature of VMM based on the properties of the VM and the available hosts. Listing 8.2 contains the code for creating the HA virtual machine.

LISTING 8.2: Creating a new highly available virtual machine

```
# Get a connection to the VMM server
$c = Get-VMMServer "localhost"

# Create the Job Group ID. This is the Guid that pairs all the cmdlets
# necessary to ensure that the new Virtual Machine creation is
# Successful. Every cmdlet that specifies the same Job Group ID will
# be part of a set and will be executed in order after the final command
# that includes the same Job Group ID runs. In this case, the final
```

```
# command is the New-VM cmdlet.
$JobGroupID = [System.Guid]::NewGuid().ToString()

# Enter the VM Name
$VMName = "virtualmachine1"

# Create a virtual NIC
New-VirtualNetworkAdapter -JobGroup $JobGroupID -PhysicalAddressType
Dynamic -VLANEnabled $false

# Create a virtual DVD
New-VirtualDVDDrive -JobGroup $JobGroupID -Bus 1 -LUN 0

# Check if another HW profile has the same name and delete it if there is
$HardwareProfile = Get-HardwareProfile | where {$_.Name -eq "HWProfile"}
if ($HardwareProfile -ne $null)
{
   Write-Warning "Deleting the existing hardware profile with the same name"
   Remove-HardwareProfile $HardwareProfile
}

# Create a new hardware profile with the user preferences
# The -HighlyAvailable property of this cmdlet is the one indicating
# that this Virtual Machine should be a Highly Available one
$HardwareProfile = New-HardwareProfile -Owner "vmmdomain\administrator"
 -Name "HWProfile" -CPUCount 1 -MemoryMB 2048 -HighlyAvailable $true
-NumLock $false -BootOrder "CD", "IdeHardDrive", "PxeBoot",
"Floppy" -LimitCPUFunctionality $false -JobGroup $VMGuid

# Create a new VHD for the VM
$DiskDrive = New-VirtualDiskDrive -IDE -Bus 0 -LUN 0 -JobGroup
$JobGroupID -Size 10240 -Dynamic -Filename "virtualmachine1.vhd"

# Get all the hosts and their ratings for this VM's HW profile
$AllHosts = Get-VMHost
$hostrating = Get-VMHostRating -VMHost $AllHosts
 -HardwareProfile $HardwareProfile -DiskSpaceGB 10 -VMName $VMName

# Order the host ratings and check if we have at least one
# positive star rating for this VM
$orderedrating = $hostrating | sort-object rating -descending
Write-Output $orderedrating

# If the rating is 0, exit and don't call new-vm
if ($orderedrating -is [Array])
{
   # we have multiple results, so pick the top one
   $targethost = $orderedrating[0].VMhost
```

```
   if ($orderedrating[0].Rating -eq 0)
   {
      Write-Warning "There is no suitable host for this VM's profile"
      Write-Warning $orderedrating[0].ZeroRatingReasonList[0]
      break
   }
}
else
{
   $targethost = $orderedrating.VMhost
   if ($orderedrating.Rating -eq 0)
   {
      Write-Warning "There is no suitable host for this VM's profile"
      Write-Warning $orderedrating.ZeroRatingReasonList[0]
      break
   }
}

Write-Output "We will be creating a new VM on host $targethost"

# Get the operating system from a list of predefined OS Names in VMM
$OperatingSystem = Get-OperatingSystem | where {$_.Name -eq
"64-bit edition of Windows Server 2008 Enterprise"}

# Find the path for the LUN/Disk to host the files for the new HA VM
# Make sure this volume is not in use, is a cluster volume
# (with cluster resources already created), and is available
# for placement

$targetpath = get-vmhostvolume -VMHost $targethost
 | where-object -filterscript {$_.IsClustered -eq $true}
 | where-object -filterscript {$_.InUse -eq $false}
 | where-object -filterscript {$_.IsAvailableForPlacement
 -eq $true}
if ($targetpath -eq $null)
{
   Write-Warning "There is no suitable cluster disk to place this VM on"
   Break
}

if ($targetpath -is [Array])
{
   # Pick the first available disk to place this VM on
   $targetpath = $targetpath[0].Name
}
else
{
   $targetpath = $targetpath.Name
}
```

```
# Create the new-vm in an asynchronous way
New-VM -VMMServer $c -Name $VMName -Description
 "new HA VM to learn more about PowerShell"
 -Owner "vmmdomain\administrator" -VMHost
 $targethost -Path $targetpath -HardwareProfile $HardwareProfile
 -JobGroup $JobGroupID -RunAsynchronously -OperatingSystem
 $OperatingSystem -RunAsSystem -StartAction NeverAutoTurnOnVM
 -StopAction SaveVM
```

P2V Conversion

VMM makes consolidating old servers a breeze with a simple wizard for converting physical servers (also known as Physical to Virtual (P2V) conversion). In this section, we will show you a couple of examples of creating a virtual machine from a given computer system. Listing 8.3 has the PowerShell code for an online P2V.

LISTING 8.3: Converting a physical server to a virtual machine without any downtime

```
# Get the connection to the VMM server
Get-VMMServer -ComputerName "localhost"

# Get the administrative credentials for accessing the source
# machine (this has to be domain credentials)
$PSCredential = Get-Credential

# Get the target host for this Virtual Machine
$VMHost = Get-VMHost -ComputerName "localhost"

# Initiate the asynchronous P2V operation
New-P2V -SourceComputerName "localhost" -VMHost $VMHost -Name
 "resultVMName" -Path $VMHost.VMPaths[0] -MemoryMB 1024 -Credential
$PSCredential -RunAsynchronously
```

To initiate an offline conversion of a Windows Server 2000 computer, or to force the offline conversion of a Windows Server 2008 computer, you need to use the `-Offline` flag of the `New-P2V` cmdlet. Listing 8.4 has the PowerShell code for an offline P2V.

LISTING 8.4: Offline conversion of a physical server to a virtual machine

```
# Get the connection to the VMM server
Get-VMMServer -ComputerName "localhost"

# Get the administrative credentials for accessing the
# source machine (this has to be domain credentials)
$PSCredential = Get-Credential
```

```
# Get the target host for this Virtual Machine
$VMHost = Get-VMHost -ComputerName "localhost"

# Create a new machine configuration for the physical source
# computer. This triggers the hardware scout to retrieve the data
# from the source computer.
New-MachineConfig -SourceComputerName "sourcemachine.vmmdomain.com"
-Credential $PSCredential
$MachineConfig = Get-MachineConfig | where {$_.Name
-eq "sourcemachine.vmmdomain.com"}

# Initiate the asynchronous offline P2V operation, using a static
# IP Address for the conversion
# If a patch file or driver is missing, download the required
# patches or driver files to the Patch Import directory on the
# VMM server (the default path is <SystemDrive>:\Program Files
# \Microsoft System Center Virtual Machine Manager 2008\
# Patch Import), and extract the files by using the
# Add-Patch cmdlet.

New-P2V -Credential $PSCredential -VMHost $VMHost -Path
 $VMHost.VMPaths[0] -Owner "vmmdomain\administrator" -Trigger
 -Name "resultVMName"
-MachineConfig $MachineConfig -Offline -Shutdown
-OfflineIPAddress "192.168.100.23" -OfflineNICMacAddress
"00:11:22:33:44:55" -OfflineDefaultGateway "192.168.100.1"
-OfflineSubnetMask "255.255.255.0" -CPUCount 1 -MemoryMB 1024
-RunAsSystem -StartAction NeverAutoTurnOnVM
 -UseHardwareAssistedVirtualization $false -StopAction SaveVM
-StartVM -RunAsynchronously
```

Virtual Machine Migrations

When you're migrating a virtual machine through Virtual Machine Manager, the transfer type (or speed) for the migration is determined during Intelligent Placement and it is based upon the properties and capabilities of the source virtual machine and the destination host along with the connectivity between them. In Listing 8.5, we attempt to migrate a virtual machine while enforcing a requirement that only SAN or cluster migrations are eligible. This puts a requirement on the script to not only check for good host ratings but also to ensure that the transfer can be accomplished quickly using a SAN or a failover cluster.

LISTING 8.5: Migrating a virtual machine using cluster or SAN and tracking the migration progress

```
# Get a connection to the VMM server
$c = Get-VMMServer "localhost"
```

```
# Get the VM to migrate
$VM = Get-VM "virtualmachine1"

# Get all the hosts
$AllHosts = Get-VMHost

# The IsMigration flag allows the current host of the VM
# to be considered as the migration target
$hostrating = Get-VMHostRating -VMHost $AllHosts -VM $VM -IsMigration

# Order the host ratings
$orderedrating = $hostrating | sort-object rating -descending
Write-Output $orderedrating

# Now search for the top rated host that can do a Cluster or a SAN migration
# All other migration options are not considered here
$targethost = $null
foreach ($rating in $orderedrating)
{
  if ($rating.Rating -gt 0)
  {
   switch ($rating.TransferType)
   {
      # These options are listed in order of decreasing transfer speed
      "Live"
         {
            Write-Output "$rating.Name has a transfer type of Live"
            $targethost = $rating.Name
            break
         }
      "Cluster"
         {
            Write-Output "$rating.Name has a transfer type of Cluster"
            $targethost = $rating.Name
            break
         }
      "San"
         {
            Write-Output "$rating.Name has a transfer type of SAN"
            $targethost = $rating.Name
            break
         }
      "Network"
         {
            Write-Output "$rating.Name has a transfer type of Network"
         }
      default
```

```
        {
           Write-Output "$rating.Name has an invalid TransferType"
        }
   }
  }

}

if ($targethost -eq $null)
{
   Write-Warning "We were not able to find a suitable destination
host for this VM with a fast transfer (SAN or Cluster)"
   break
}

# Migrate the VM to the target host
$VMHost = Get-VMHost -ComputerName $targethost
$resultvm = Move-VM -VM $VM -vmhost $VMHost -Path $VMHost.VMPaths[0]
-RunAsynchronously

# Get the VMM Job that was launched for this migration
$job = $resultvm.MostRecentTask

# Iterate the loop until the Job is finished while reporting progress
while ($job.Status -eq "Running")
{
   $progress = $job.Progress
   Write-Progress $VM Progress -PercentComplete $job.ProgressValue -ID 1
   Start-Sleep 3
}

# The VMM job is now finished (either with a failed or a completed status)
$status = $job.Status
Write-Warning "Migration of $VM to host $VMHost finished
with a status of: $status"
$error = $job.ErrorInfo | select DisplayableErrorCode,
Problem, RecommendedActionCLI
Write-Warning $error
```

In some cases, it is beneficial to force a LAN migration even when a faster migration option is available. The only change from the code in Listing 8.5 would be a small change in the `Move-VM` cmdlet to add the `UseLAN` option as indicated here.

```
# Use the -UseLAN option to force a Network transfer of the Virtual Machine
$resultvm = Move-VM -VM $VM -vmhost $VMHost -Path $VMHost.VMPaths[0]
-RunAsynchronously -UseLAN
```

Provisioning Multiple Virtual Machines

Virtual machines are usually provisioned on demand based on customer requirements. However, there are cases where having many virtual machines available for immediate use is a requirement. Such scenarios might include hosted desktops allocating virtual machines from a pool of VMs or allocating VMs to an enterprise application based on load. In the PowerShell script in Listing 8.6, we read the input from a text file and create virtual machines in blocks of five at a time (the throttling rate is customizable). This allows us to customize and repeat the automated provisioning process by updating a text file rather than having to adjust a PowerShell script.

LISTING 8.6: PowerShell script for creating multiple VMs based on an input file

```
# get the command line arguments passed to this script
$length = $args.length
$expectedArgsLength = 2

# The script takes as input the customization filepath and the VMM server name
$usage = "Usage: ScriptName.ps1 <customizationfile.txt> <vmm-server-name>"
if ($length -ne $expectedArgsLength)
{
   write-warning $usage;
   break
}

# The ArrayList to use for tracking new-vm creations
$arraylist = New-Object System.Collections.ArrayList
$arraylist.Clear()

# The max number of concurrent VM creations (throttling rate)
$MaxCreations = 5

# Get a connection to the VMM server
$servername = $args[1]
get-vmmserver -ComputerName $servername

# now open the customization file to read its input
$customFile = $args[0]
$content = get-content $customFile
foreach ($values in $content)
{
   # $values contains one line of input. Each line represents a VM
   # now split the CSV input line
   $newvalues = $values |% {$_.split(",")}

   # Perform a test to ensure the proper number of parameters exist
   if ($newvalues.length -ne 14)
```

```
   {
      write-warning "The proper number of parameters does not exist for $values";
      break
   }

   # get the input variables from the file and into the specific variables
   $vmname = $newvalues[0]            # The virtual machine name
   $computername = $newvalues[1]      # The guest OS computer name
   $memory = $newvalues[2]            # The amount of RAM to allocate to the VM
   $OSSKU = $newvalues[3]             # The OS name (VMM has these
already defined)
   $ProductID = $newvalues[4]         # The Windows Product ID
   $description = $newvalues[5]       # A description for the VM
   $vmpath = $newvalues[6]            # The path where to create this VM
   $vnetworkname = $newvalues[7]      # The Virtual Network Name
   $hostname = $newvalues[8]          # The name of the host to place this VM on
   $cpuvalue = $newvalues[9]          # The CPU Name (VMM has these
already defined)
   $cpucount = $newvalues[10]         # The number of CPUs
   $owner = $newvalues[11]            # The owner of the VM
   $adminpwd = $newvalues[12]         # The guest OS administrator password
   $templatename = $newvalues[13]     # The template name from
which to create this VM

   # Create the Job Group ID and the hardware profile name
   $jobguid = [guid]::NewGuid().ToString()
   $profilename = "Profile" + $jobguid

   # create the VM based on the settings in the file - this will
happen asynchronously
   Set-VirtualFloppyDrive -RunAsynchronously -VMMServer $servername
-NoMedia -JobGroup $jobguid
   New-VirtualNetworkAdapter -VMMServer $servername -JobGroup
 $jobguid  -PhysicalAddressType Dynamic -VirtualNetwork $vnetworkname
-VLanEnabled $false
   New-VirtualDVDDrive -VMMServer $servername -JobGroup $jobguid  -Bus 1 -LUN 0
   $CPUType = Get-CPUType -VMMServer $servername | where {$_.Name -eq $cpuvalue}
   New-HardwareProfile -VMMServer $servername -Owner $owner
-CPUType $CPUType -Name $profilename -Description "Profile used to
create a VM/Template" -CPUCount $cpucount -MemoryMB
$memory -ExpectedCPUUtilization 20 -DiskIO 0 -NetworkUtilization
10 -RelativeWeight 100 -HighlyAvailable $false -NumLock $false
-BootOrder "CD", "IdeHardDrive", "PxeBoot", "Floppy"
-LimitCPUFunctionality $false -JobGroup $jobguid
   $Template = Get-Template -VMMServer $servername | where
{$_.Name -eq $templatename}
   $VMHost = Get-VMHost -VMMServer $servername | where {$_.Name -eq $hostname}
   $HardwareProfile = Get-HardwareProfile -VMMServer localhost |
```

```
where {$_.Name -eq $profilename}
   $OperatingSystem = Get-OperatingSystem -VMMServer localhost |
where {$_.Name -eq $OSSKU}

   # Before we start the new-vm creation we need to check
   # if we reached the maximum number of concurrent creations
   while ($arraylist.Count -eq $MaxCreations)
   {
      $toremove = $null
      foreach ($jobid in $arraylist)
      {
         # get the current status of the job
         $tempjobid = [string]::join("", $jobid.Keys)
         $tempjob = Get-Job -ID $tempjobid;
         if ($tempjob.Status -ne "Running")
         {
            # This job completed, so remove it from the tracking list
 so that new VMs can be created
            Write-Output "Job $tempjobid finished running"
            $toremove = $jobid
            break
         }
      }

      if ($toremove -ne $null)
      {
         $arraylist.Remove($jobid)
      }

      Start-Sleep 2
   }

   # if we reached here, it is safe to create the new VM
   $resultvm = New-VM -Template $Template -Name $vmname
 -Description $description -VMHost $VMHost -Path $vmpath -JobGroup
 $jobguid -Owner $owner -HardwareProfile $HardwareProfile
 -ComputerName $computername -FullName "" -OrgName "" -ProductKey
 $ProductID -TimeZone 4 -JoinWorkgroup "WORKGROUP" -OperatingSystem
 $OperatingSystem -RunAsSystem -StartAction
 NeverAutoTurnOnVM -UseHardwareAssistedVirtualization $false
 -StopAction SaveVM -RunAsynchronously

   # Now start tracking this new-vm instance
   if ($resultvm -ne $null)
   {
      # Get the VMM Job that was launched for this migration
```

```
        $job = $resultvm.MostRecentTask
        $arraylist.Add(@{$job.ID = $job})
    }
}

write-output "Done creating All VMs!"
```

The following code contains a sample line from an input text file that can be used in the script in Listing 8.6. This line contains the values for the different virtual machine properties that are needed by the PowerShell script. These values need to be specified in order, and their descriptions are as follows:

1. The virtual machine name
2. The guest OS computer name
3. The amount of RAM or memory to allocate to the VM
4. The OS name (VMM has these already defined)
5. The Windows product ID
6. A description for the VM
7. The path describing where to create this VM
8. The virtual network name
9. The name of the host on which to place this VM
10. The CPU name (VMM has these already defined.)
11. The number of CPUs
12. The owner of the VM
13. The guest OS administrator password
14. The name of the template from which to create this VM

```
vmname1,vmname1ComputerName,1024,64-bit edition of Windows Server 2008
 Enterprise,55555-55555-55555-55555-55555,scripted VM,D:\ProgramData\Microsoft
\Windows\Hyper-V,Broadcom NetXtreme 57xx Gigabit Controller - Virtual
 Network,hypervhost1.vmmdomain.com,2.40 GHz Xeon,1,vmmdomain\administrator,
password,MyTemplate
```

Automating the Addition of Managed Hosts

Adding a virtual machine host to VMM requires administrative credentials for the physical computer. The requirement of credentials makes it hard to automate any tasks that need to run unattended without sacrificing the security of your credentials. Since the credentials are required parameters to the VMM cmdlets in the script and storing them in clear text might

compromise security, you can save your credentials to a file for later use. See the following PowerShell cmdlets for an example:

```
# First, call Get-Credential to store your credentials to a PowerShell variable
$PSCredential = Get-Credential

# Now construct the file path of the file that will store the credentials
$SecureFilePath = $PSCredential.UserName + ".cred"
$SecureFilePath = $SecureFilePath.Replace("\", "_")

# Store the credentials to this file
$PSCredential.Password | ConvertFrom-SecureString | Set-Content $SecureFilePath
```

When it is time to execute the automated task, you can retrieve this file from the same location and add a host to VMM. In the following code snippet, we are looking to add all Hyper-V hosts in the environment:

```
# Get the password from the file that we used earlier to store it in
$Password = Get-Content $SecureFilePath | ConvertTo-SecureString

# Create a new PsCredential object for our administrator
# using the stored password
$PSCredential_Out = New-Object
System.Management.Automation.PsCredential("vmmdomain\administrator",
$Password)

# Now discover all the Hyper-V hosts in the domain whose name starts with HyperV
$Computers = Discover-Computer -ComputerNameFilter "HyperV" -Domain
"vmmdomain.com" -FindHyperVHosts -ExcludeVMMHost | select Name

# The output of the discover-computer cmdlet can now be used to
# add these hosts to VMM
foreach ($computer in $Computers)
{
   # Instead of prompting for credentials, $PSCredential_Out contains the values
   # required by VMM to add a new host
   Add-VMHost -Credential $PSCredential_Out -ComputerName $computer
}
```

Working with MAC Addresses

Virtual Machine Manager manages a static range of MAC addresses that can be used when attaching a virtual network device to a Virtual Machine. MAC addresses that are consumed from this static range can never be reused, in the same way the MAC addresses on physical machines are all unique. To configure the MAC address range, click on the administration view of the VMM Administrator Console and select the Networking option. Figure 8.14 shows a sample MAC address range for a VMM deployment.

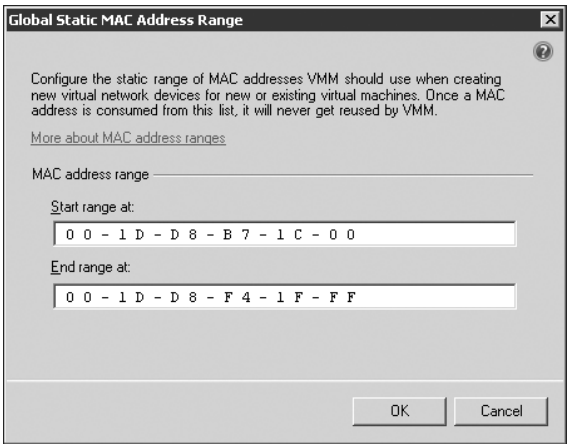


FIGURE 8.14 Global Static MAC Address Range dialog box

Listing 8.7 shows how to get the next available MAC address from this range and how to commit the selection. Once the MAC address is committed, it will never be used again by VMM.

LISTING 8.7: PowerShell script to retrieve the next available MAC address

```
# First, invoke the New-PhysicalAddress cmdlet to view the next
# available MAC address in the range
# Output will be something like this: 00:1D:D8:B7:1C:00
New-PhysicalAddress

# if you execute the New-PhysicalAddress multiple times, the
# output will not change from 00:1D:D8:B7:1C:00
New-PhysicalAddress

# Now, let's save this MAC address and commit the change in VMM
$MacAddress = New-PhysicalAddress -Commit

# Print the MAC address we just committed (should be 00:1D:D8:B7:1C:00)
$MacAddress

# Show that the next invocation of New-PhysicalAddress will
# return a new MAC Address from the range. (00:1D:D8:B7:1C:01)
New-PhysicalAddress

# Once you have a MAC address, you can invoke the Set-VirtualNetworkAdapter
# cmdlet to set the MAC Address
# First, let's get the virtual network adapter for our Virtual Machine1
$vnic = get-virtualnetworkadapter -vm "<insert Virtual Machine name>"
```

```
# Now set the properties of the adapter to include this static MAC Address
$vnic | Set-virtualnetworkadapter -Physicaladdresstype Static
-PhysicalAddress $MacAddress
```

Evacuating a Host for Maintenance

It is sometimes necessary to service the physical computer running the virtualization software, resulting in several hours of downtime. We will show you a script that you can use to evacuate a host from all of its virtual machines instead of the virtual machines being inactive as well. Make sure you monitor the progress of the VMM jobs to ensure that all virtual machines have successfully migrated to a different host. Listing 8.8 contains the code for asynchronously moving all the VMs from a host.

VMM 2008 R2 also introduced a new feature called maintenance mode. A host managed by VMM can be placed into maintenance mode if you want to perform maintenance tasks on the physical host (e.g., replace hardware or install security updates that might require a server reboot). Once a host is in maintenance mode, VMM will no longer allow that host to be the target host for a new virtual machine. In addition, a host that is in maintenance mode is excluded from host ratings calculations during virtual machine placement.

When maintenance mode is initiated on a host, all running virtual machines are put into a saved state. If the host is part of a cluster, then the user is presented with the option to either live migrate all its virtual machines to another host or to save the state of all virtual machines on that host. Live migration is an option only if the host cluster is capable of live migration. This behavior is a little bit different for VMware hosts. Once a VMware ESX host is put into maintenance mode in VMM, VMM will send an ''Enter maintenance mode'' request to VMware Virtual Center. The behavior of the VMs on that host is determined based on the configuration of the maintenance mode feature in Virtual Center.

When maintenance mode is stopped on a host, VMM will allow that host to be the target host of migrations and that host will start receiving a star rating in placement calculations. However, no VMs are restarted on that host, and the VMs that were migrated away from that host are not placed back automatically.

When you're using the maintenance mode feature of VMM 2008 R2, the `Disable-VMHost` cmdlet places a virtual machine host into maintenance mode while `Enable-VMHost` removes a host from maintenance mode.

LISTING 8.8: PowerShell script to asynchronously move all the VMs from a given host

```
# get the command line arguments passed to this script
$argslength = $args.length
$expectedArgsLength = 2

# The script takes as input the VMM server name and
# the FQDN of the host to evacuate
$usage = "Usage: ScriptName.ps1 <vmm-server-name> <Host FQDN>"
if ($argslength -ne $expectedArgsLength)
{
   write-warning $usage; break
}
```

```
# helper function to move a VM to the host with the highest star rating
# This function could be easily modified to only move VMs within a
SAN or a cluster
function MoveVM($vmobj, $hostobj)
{
   $hostrating = get-vmhostrating -vmhost $hostobj -vm $vmobj
   $orderedrating = $hostrating | sort-object rating -descending
   Write-Output $orderedrating

   $targethost = $null
   if ($orderedrating -is [Array])
   {
      if ($orderedrating[0].Rating -ne 0)
      {
         $targethost = $orderedrating[0].VMhost
      }
   }
   else
   {
      if ($orderedrating.Rating -ne 0)
      {
         $targethost = $orderedrating.VMHost
      }
   }

   if ($targethost -ne $null)
   {
      write-warning "Moving VM $vmobj to host $targethost"
      $resultvm = move-vm -VM $vmobj -vmhost $targethost
-Path $targethost.VMPaths[0] -RunAsynchronously
   }
   else
   {
      Write-Warning "There is no suitable host for this VM $vmobj
and it will not be migrated!"
   }
}

# get a connection to the VMM server
$vmmserver = $args[0]
$c = get-vmmserver -ComputerName $vmmserver

# Now call Get-VM to cache all the VMs in Powershell
$vms = Get-VM

# Get the host computer and all hosts
$hostname = $args[1]
$VMHost = Get-VMHost -ComputerName $hostname
$AllHosts = Get-VMHost
```

```
# Now set this host to maintenance mode to prevent VMs from
# being deployed here
$VMHost | Set-VMHost -MaintenanceHost $true

# Enumerate all VMs on this host and move them asynchronously
foreach ($VM in $VMHost.VMs)
{
   MoveVM $VM $AllHosts
}
```

Utilizing Rapid Provisioning

VMM 2008 R2 introduced a new feature called Rapid Provisioning. This feature was implemented in response to customer demand to improve the time required to create virtual machines. In VMM 2008, the only way to create and deploy a new virtual machine was by utilizing a template, another virtual machine, or a VHD from the VMM library. During the new virtual machine creation process, VMM copied all the required VHDs over the network using the BITS protocol. Depending on the size of VHD and the available bandwidth, this operation could take several minutes to complete.

Several customers have sophisticated SAN technologies that enable them to clone a LUN that contains the VHD and present it to the host. However, customers still want to leverage the VMM template capabilities with operating system (OS) customization. Rapid Provisioning allows you to take advantage of your fast SAN infrastructure to move (or copy) the actual VHD files to the host but tie that back to VMM's rich template customization process. With Rapid Provisioning, you can now create a template that includes the OS configuration and references a ''dummy'' blank VHD. The blank VHD will not be used and will be replaced through the `Move-VirtualHardDisk` cmdlet. This cmdlet will let VMM know that it should not be using the VHD that is referenced in the template. Instead, VMM should use a VHD that resides locally on the host computer. To indicate to VMM that Rapid Provisioning needs to be used, the `New-VM` cmdlet takes a new switch called `UseLocalVirtualHardDisk`. Rapid Provisioning is only available through Windows PowerShell cmdlets.

Listing 8.9 shows an example creation of a virtual machine by utilizing Rapid Provisioning. In this example, `C:\Win2k8_Base_OS_Sysprep.vhd` has to locally exist on the host computer before the `New-VM` cmdlet is invoked.

LISTING 8.9: Creating a new virtual machine using Rapid Provisioning

```
# Start by specifying the file location for the VHD that will
# be used by the Virtual Machine
$VHDName = "c:\Win2k8_Base_OS_Sysprep.vhd"

# Specify other variables for new-vm cmdlet
$vmname = "vm1"
$hostname = "host.contoso.com"
```

```
# Get an instance of the host that will be the target
# for the Virtual Machine
$vmhost = get-vmhost $hostname

# Create the jobgroup ID for new-vm from template
$JobGuid = [System.Guid]::NewGuid().ToString()

# Specify the local location for the VHD
# That will replace the "dummy" VHD that exists in the template
# VMM expects that $VHDName already exists on the host computer
# when the new-vm cmdlet is called.
Move-VirtualHardDisk -Bus 0 -LUN 0 -IDE -Path $VHDName -JobGroup $JobGuid

# Get the instance of the template that will be used for OS Configuration
$template = Get-Template | where {$_.Name -eq "VMMTemplate"}

# Get the current username to be passed as the Virtual Machine owner
$callerUsername = whoami

# Create the new-vm from template and specify the Rapid
# Provisioning flag (-uselocalvirtualharddisks)
New-VM -Template $template -Name $vmname -Description
"a Virtual Machine created with RP" -Owner $callerUsername
-VMHost $vmhost -UseLocalVirtualHardDisks -Path $vmhost.VMPaths[0]
-RunAsynchronously -JobGroup $JobGuid | Out-Null
```

Even though Virtual Machine Manager does not provide UI support for creating a virtual machine using differencing VHD disks, this can be accomplished using Rapid Provisioning. Using the public Hyper-V WMI interface, you can create a differencing disk for the `c:\Win2k8_Base_OS_Sysprep.vhd` VHD file used in Listing 8.9. Then, when the `Move-VirtualHardDisk` cmdlet is executed, you can pass the full path to the child VHD. The differencing disk will then be used as the target VHD for the new virtual machine creation. Such a process would make it easy for customers to copy a single base disk with the operating system on a host and then use the Rapid Provisioning feature to create multiple virtual machines using differencing disks off that same parent VHD. The following code snippet shows you a partial script that creates a differencing disk from the base VHD. Then it supplies the new VHD file path to VMM for Rapid Provisioning. The following code can be used within Listing 8.9 to create a new virtual machine using differencing disks and Rapid Provisioning:

```
# Get the Image Management Service WMI instance for the host computer
$VHDService = get-wmiobject -class "Msvm_ImageManagementService"
-namespace "root\virtualization" -computername $hostname

# Create a differencing disk from the base disk
$DiffVHDName = "c:\Win2k8_Base_OS_Sysprep_child.vhd"
$Result = $VHDService.CreateDifferencingVirtualHardDisk($DiffVHDName, $VHDName)
```

```
# Wait until the Hyper-V differencing disk creation is complete
# and then pass DiffVHDName to the Move-VirtualHardDisk cmdlet
# This will notify VMM to use the differencing disk for New-VM
# instead of the base disk $VHDName
Move-VirtualHardDisk -Bus 0 -LUN 0 -IDE -Path $DiffVHDName
-JobGroup $JobGuid
```

In addition to the new `UseLocalVirtualHardDisks`, VMM 2008 R2 has added one more new switch for `New-VM` called `SkipInstallVirtualizationGuestServices`. This switch notifies VMM to skip the installation of the Integration Components (ICs) (also known as Virtual Guest Services) as part of the `New-VM` cmdlets, decreasing the amount of time required for `New-VM` to complete. This switch should be used only if you are already certain that your template either contains the ICs or contains an operating system that has built-in integration components. It is important that you ensure that all your VMs have the integration components correctly installed to take full advantage of virtualization and virtualization management. The `SkipInstallVirtualizationGuestServices` will take effect only in the following three `New-VM` scenarios:

- New virtual machine from VHD
- New virtual machine utilizing an existing virtual machine
- New virtual machine from a template that does not have an OS configuration specified

The following code shows an example invocation of the `New-VM` cmdlet that utilizes this new switch. This new switch can also be used along with the `UseLocalVirtualHardDisks` switch to further speed up the `New-VM` process. Here's the code:

```
# Specify variables needed for the new-vm cmdlet
$vmname = "vm2"
$hostname = "host.contoso.com"

# Get an instance of the host that will be the target
# for the Virtual Machine
$vmhost = get-vmhost $hostname

# Create the jobgroup ID for new-vm from template
$JobGuid = [System.Guid]::NewGuid().ToString()

# Get the instance of the template that will be used for OS Configuration
$template = Get-Template | where {$_.Name -eq "VMMTemplate"}

# Get the current username to be passed as the Virtual Machine owner
$callerUsername = whoami

# Create the new-vm from template and specify the
# SkipInstallVirtualizationGuestServices switch to skip
# the Install VM components step of the New-VM cmdlet
New-VM -Template $template -Name $vmname -Description
"a Virtual Machine created with RP" -Owner $callerUsername
```

```
-VMHost $vmhost -SkipInstallVirtualizationGuestServices
-Path $vmhost.VMPaths[0] -RunAsynchronously
-JobGroup $JobGuid | Out-Null
```

Specifying CPU Settings

The Virtual Machine Manager Administrator Console only exposes the Virtual Machine Priority setting as part of the virtual machine properties. The priority of a VM, which decides how to allocate CPU resources on the host for this VM, can be specified in the Hardware Configuration tab, as shown in Figure 8.15. VMM exposes two more CPU properties for a virtual machine through Windows PowerShell only and the `Set-VM` cmdlet:

`CPUMax` Specifies the highest percentage of the total resources of a single CPU on the host that can be used by a specific virtual machine at any given time.

`CPUReserve` Specifies the minimum percentage of the resources of a single CPU on the host to allocate to a virtual machine. The percentage of CPU capacity that is available to the virtual machine is never less than this percentage.

A third PowerShell property, called `RelativeWeight`, is the same as the VM Priority property seen in Figure 8.15. Use this command to create a new hardware profile with the three CPU properties set at different levels.

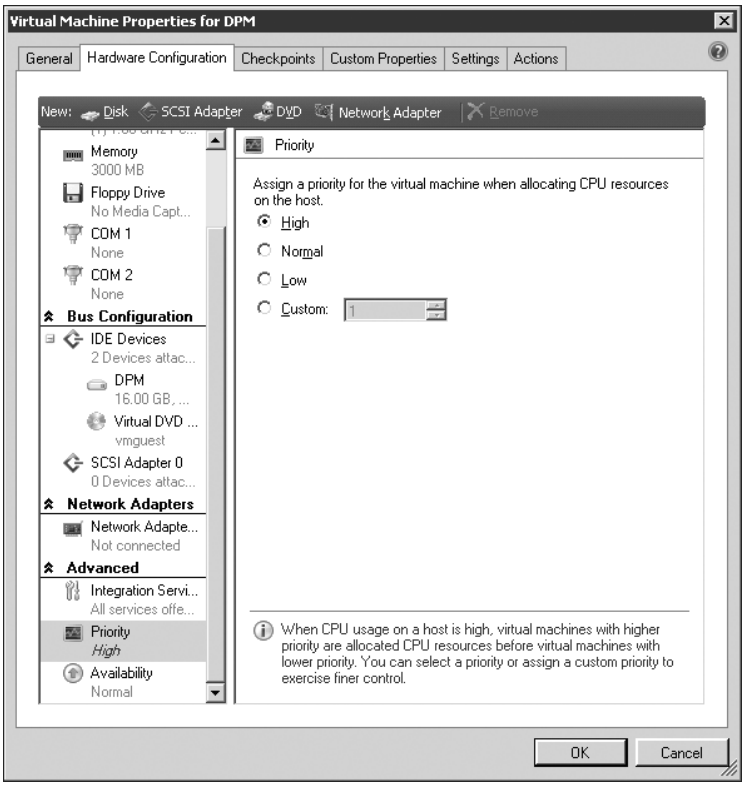


FIGURE 8.15
CPU priority for a virtual machine

The following code shows an example creation of a virtual machine with the CPU properties:

```
# Get the instance of a host
$vmhost = get-vmhost "host.contoso.com"

# Get the instance of a VHD that will be used during New-VM
$vhd = (Get-VirtualHardDisk)[0]

# Create a new hardware profile with the CPU settings set
$hwProfile = New-HardwareProfile -Name "cpuHWProfile" -description ""
 -CPUMax 70 -CPUReserve 50 -RelativeWeight 80

# Create the new Virtual Machine with the hardware profile specified
New-VM -Name "cpuVM" -VirtualHardDisk $vhd -VMHost $vmhost -HardwareProfile
 $hwProfile -Path $vmhost.VMPaths[0]

# Show that the new Virtual Machine created has the specified CPU settings
Get-VM -Name "cpuVM" | Select Name, Hostname, CPUMax, CPUReserve,
 RelativeWeight
```

Clustering Cmdlet Switches

VMM 2008 R2 provides support for the new Windows Server 2008 R2 Hyper-V features, including Live Migration in a failover cluster environment. From the VMM Administrator Console, if Live Migration is available for a virtual machine, then that is the only option offered to an administrator for migrating the virtual machine to another node in the cluster. If you want to force the transfer type of the virtual machine to be Quick Migration (Quick Migration saves the state of a virtual machine prior to changing its ownership to another node in the cluster) even if Hyper-V Live Migration is available, use the `UseCluster` switch with this command:

```
Move-VM -VM $myVM -vmhost $VMHost -Path $VMHost.VMPaths[0] -UseCluster
```

Hyper-V Live Migration allows only one cluster node to participate in a live migration at any point in time. VMM implemented a queue to track active live migrations and ensure that all user-executed live migrations complete in order. If you would like the `Move-VM` cmdlet to fail if a Hyper-V live migration is in progress and your live migration cannot start immediately, use the `BlockLMIfHostBusy` switch with this command (this switch will not utilize the VMM Live Migration queue):

```
Move-VM -VM $myVM -vmhost $VMHost -Path $VMHost.VMPaths[0] -BlockLMIfHostBusy
```

Monitoring and Reporting

Creating automated tasks that checks the health of your system and emails the administrator on critical errors can be accomplished very easily with a few cmdlets. In this section, we will show you a few cmdlets that can prove useful in assessing the overall health of your system. If you would like to bundle these scripts into an automated task and enable email notification, you can look into the SMTP emailing properties of the class `System.Net.Mail.MailMessage`.

Use this command to get the overall status of the virtual machines' health:

```
Get-VM | Select Name, ID, Status | sort-object Status
```

Use this command to get a list of all virtual machines and their host names:

```
# VMID is the unique identifier of the VM on the virtualization platform
(i.e. Hyper-V)
# ID is the unique identifier of the VM in Virtual Machine Manager
Get-VM | Select Name, ID, HostName, VMID
```

Use this command to get a list of the last job run on each virtual machine:

```
Get-VM | Select Name, ID, MostRecentTask
```

Use this command to get the health information of the hosts:

```
Get-VMHost | select Name, OverallState, CommunicationState,
VirtualServerState, VirtualServerVersionState
```

Use this command to get the health information of the managed physical computers:

```
Get-VMMManagedComputer | select Name, State, VersionState, UpdatedDate
```

Use this command to create a report of two custom properties of a virtual machine:

```
# You can use the Customer Properties of a VM to add any data you would like to
# associate with a VM. In this example, we chose CostCenter and
# LastUpdated for the first two custom properties
Get-VM |select Name, Status,
 @{Name='CostCenter';Expression={$_.CustomProperties[0]}},
 @{Name='LastUpdated';Expression={$_.CustomProperties[1]}}
```

Use this command to get the list of virtualization platforms in VMM:

```
Get-VMHost | select Name, VirtualizationPlatformDetail | sort-object
 VirtualizationPlatformDetail -descending
```

Use this command to get the last 10 jobs that were run in VMM, their owners, and the affected objects:

```
$jobs = get-job | sort-object StartTime -Descending | select Name,
ResultName, ResultObjectType, Status, Owner
$jobs[0..10]
```

When invoking Windows PowerShell cmdlets, it is useful to be able to identify if an error occurred. Use the following command to clear any existing errors in the error pipeline and then use the same object to check for errors:

```
# Clear any existing errors from the error pipeline
$Error.Clear()
```

```
# Invoke a cmdlet. As an example, I used get-vmmserver
$c = get-vmmserver -ComputerName "localhost"

# Check if any errors occurred
if ($Error.Count -ne 0)
{
    # An error occurred here. Do something about it and terminate
    # the script
}
```

The Bottom Line

Describe the main benefits that PowerShell offers for VMM. Windows PowerShell is a relatively new technology that was developed by Microsoft Corporation. Virtual Machine Manager utilized this technology as the scripting public API for VMM and as the backbone of the Administrator Console.

> **Master It** What version of Windows PowerShell does VMM support?
> Which are the VMM assemblies needed for programmatically integrating with VMM's cmdlets?
> List the benefits that Windows PowerShell cmdlets offer as a public API.

Create scheduled PowerShell scripts. Scheduling PowerShell scripts allows an administrator to perform operations during nonwork hours and get reports on the progress and the results of those operations.

> **Master It** How can you create a scheduled task in Windows?
> List an example PowerShell script that checks if any host is in an unhealthy state and needs an administrator to take a look at it.

Use the VMM PowerShell cmdlets. Understanding the usage, scope, and association of the different VMM cmdlets and PowerShell objects allows an administrator to effectively manage VMM through Windows PowerShell.

> **Master It** How can you identify the proper parameters and syntax for the `Add-VMHost` cmdlet?
> How can you add the VMM PowerShell snap-in programmatically to a PowerShell script?
> How does the Windows PowerShell pipeline work?