Chapter 5. CustomSettings.ini and Task Sequence

In the previous chapter, we discussed some tips and tricks to customize the default user profile via policy, script, halting the MDT process mid-capture and modifying it manually, and so on.

In this chapter, we will review the essence of deployment, the task sequence that is called to perform the automated deployment in MDT. We will discuss the following topics:

- We will discuss the `CustomSettings.ini` file as it relates to the `Unattend.xml` file, what it does, how it works, and how to use it for your environment
- We'll also get into the subject of the task sequence, the scripting environment, logging, and how to use existing, as well as custom variables

The structure of the CustomSettings.ini file

As we look at the structure of the **Rules** tab of a share, we are also looking at the `CustomSettings.ini` file. The structure of the file is fairly basic; there are two sections, the `[Settings]` area and the `[Default]` area:

```
[Settings]
Priority=Default
Properties=MyCustomProperty

[Default]
OSInstall=Y
SkipAppsOnUpgrade=YES
SkipCapture=NO
SkipAdminPassword=YES
SkipProductKey=YES
_SMSTSOrgName=MDT Reference Task Sequence
SkipBitLocker=YES
SkipDomainMembership=YES
JoinWorkgroup=Workgroup
```

```
SkipFinalSummary=YES
SkipLocaleSelection=YES
SkipSummary=YES
SkipTimeZone=YES
SkipUserData=YES
TimeZoneName=Eastern Standard Time
```

Now what we see is under `[Settings]`; there are two entries, `Priority` and `Properties`. `Priority` defines the grouping order that is to be followed when there is a settings conflict. `Properties` is a place to define custom properties, which we will discuss later in this chapter.

So, on `Priority`, note that we only have one section, and its labelled `[Default]`. An example of how we might do this differently than default would be, perhaps, if we wanted to dynamically name workstations or laptops, based on chassis types.

However, let's walk through this as if we just stumbled upon this problem to show the workflow. First, we'd want to designate the machine name as something unique. Every machine has a unique serial number (we hope), so let's use that as follows:

```
[Default]
OSDComputerName=%serialnumber%
```

Well fair enough, we have it set now, so the machine names are unique, but we don't know if it's a laptop or desktop, do we?

For this, some would say we need the MDT Database SQL backend, but that's not really true:

```
[Settings]
Priority=IsLaptop, Default
Properties=MyCustomProperty

[IsLaptop]
Subsection=Laptop-%IsLaptop%
[Laptop-True]
OSDComputerName=L%serialnumber%

[Laptop-False]
OSDComputerName=D%serialnumber%

[Default]
OSInstall=Y
```

The preceding code will specify, based on the `variables.dat` readout of the chassis type, the computer name at the time of setup. Therefore, the `%serialnumber%` variable is queried and placed in `OSDComputerName` as either an `L` or `D` followed by the variable (a mixture of alphanumeric characters assumedly).

Some organizations will place laptops and desktops into different organizational units in Active Directory as well, and we can accommodate this here quite easily as well:

```
[Settings]
Priority=IsLaptop, Default
Properties=MyCustomProperty

[IsLaptop]
Subsection=Laptop-%IsLaptop%
[Laptop-True]
OSDComputerName=L%serialnumber%
MachineObjectOU=OU= Laptops,OU= User
Computers,DC=domain,DC=com

[Laptop-False]
OSDComputerName=D%serialnumber%
MachineObjectOU=OU= Desktops,OU= User
Computers,DC=domain,DC=com

[Default]
OSInstall=Y
```

What if the serial number exceeds 15 characters (and the **Network Basic Input/Output System** (**NetBIOS**) name)? No problem! Remember this is a scripting environment. We can modify the preceding snippet to account for the excessive characters by changing `OSDComputerName` to `L#Right("%serialnumber%",8)#`:

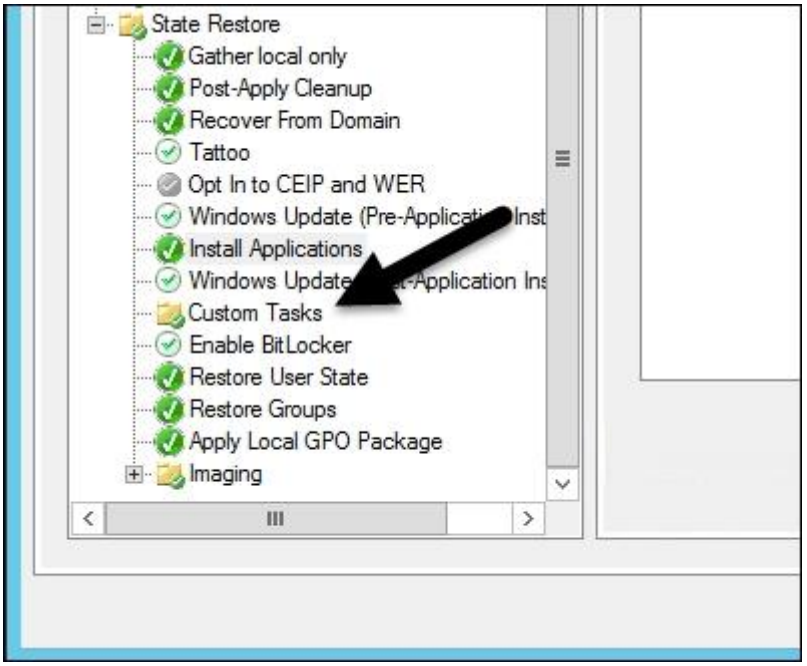
```
[Settings]
Priority=IsLaptop, Default
Properties=MyCustomProperty

[IsLaptop]
Subsection=Laptop-%IsLaptop%
[Laptop-True]
OSDComputerName=L#Right("%serialnumber%",8)#
```

```
MachineObjectOU=OU= Laptops,OU= User
Computers,DC=domain,DC=com

[Laptop-False]
OSDComputerName=D#Right("%serialnumber%",8)#
MachineObjectOU=OU= Desktops,OU= User
Computers,DC=domain,DC=com

[Default]
OSInstall=Y
```

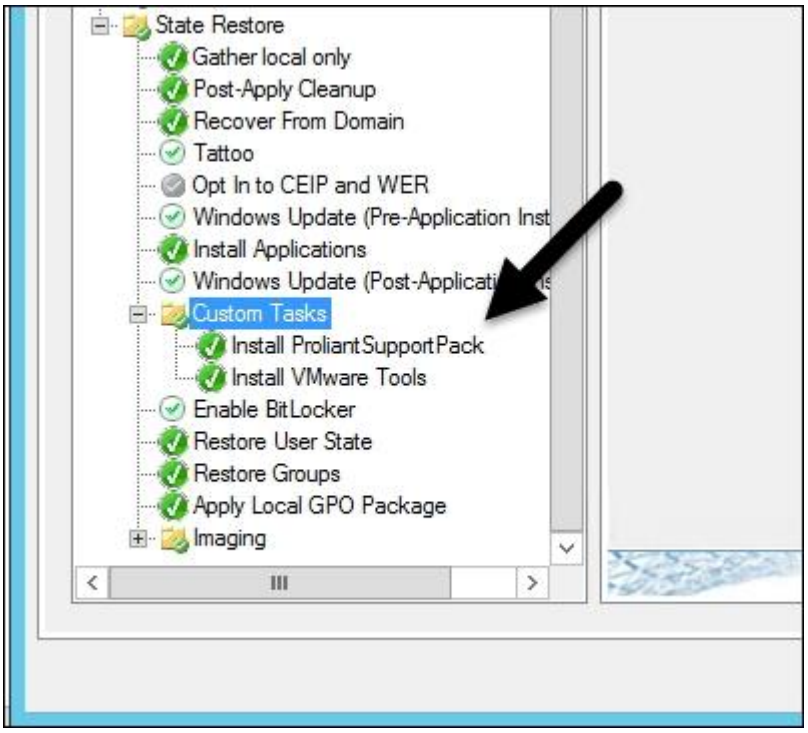
Now our `OSDComputerName` is a combination of `L` (or `D`) and then the right-most eight characters of the serial number. Now, the name is within the 15-character requirement of NetBIOS.

This same construct of logic can be used in the GUI as well! I will use the following example for this: what if we want to install a set of applications, but we will only install it if it is a VM, or only if it's a particular model number of hardware? For this example, let's assume that we are deploying to HP Servers and VMware VMs and we want to install VMware Tools, only if it is a VM, and the ProliantSupportPack software from HP, only if the task sequence is running on physical machines. Easy! The steps to do so are as follows:

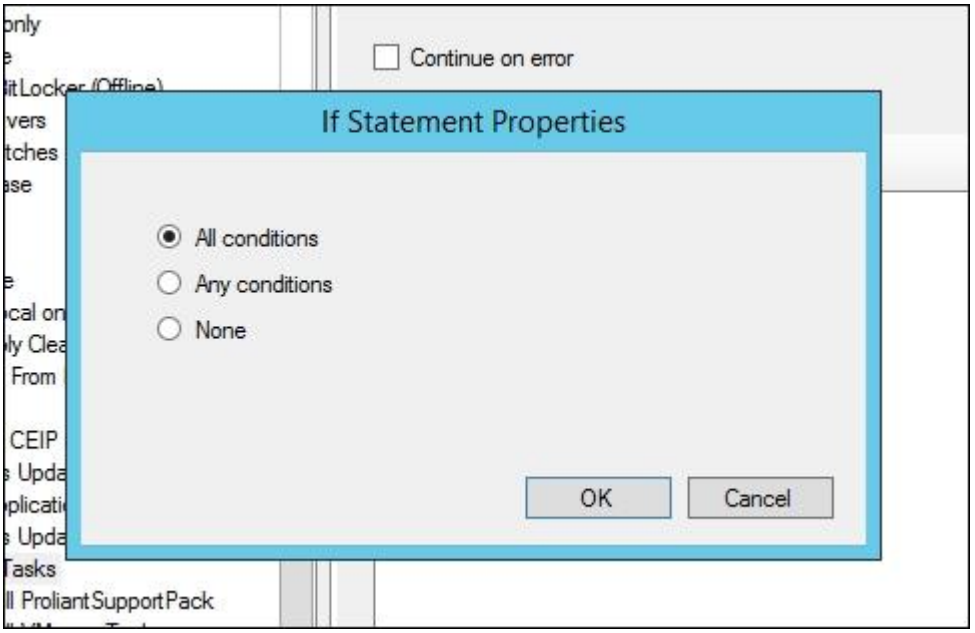
1. Import the applications for each, and then create an application step for each under **Custom Tasks**:



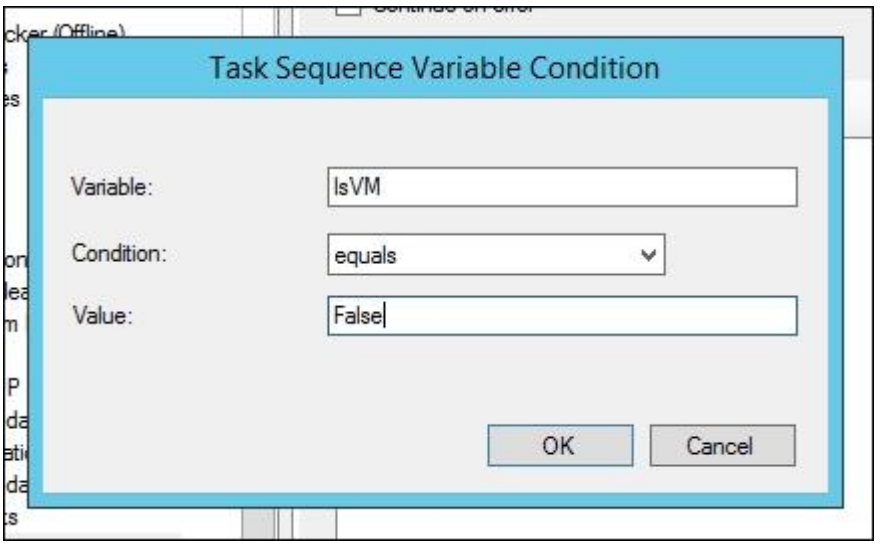
It will appear as shown in the following screenshot:



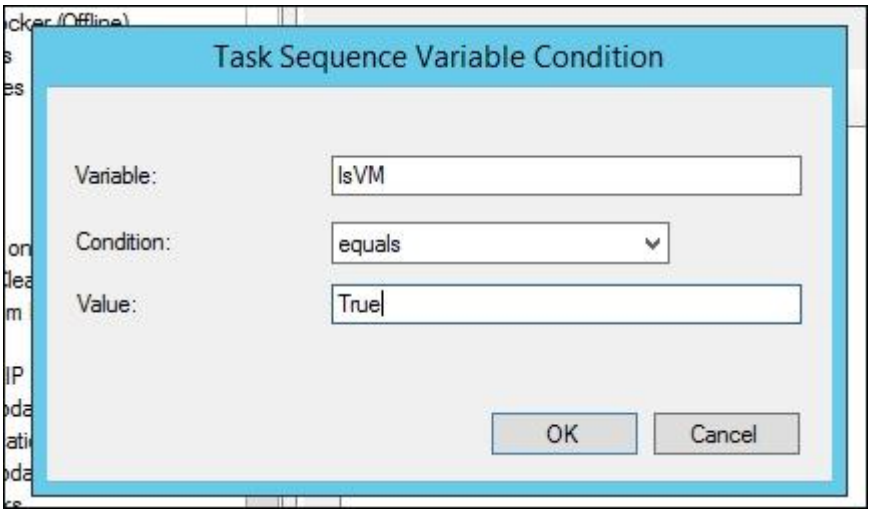
2. Then, in the **Properties** of each step, in the **Options** tab, add an **If Statement Properties** condition, as shown in the following screenshot:



3. Then simply set the condition for physical machine step as follows (**Install ProliantSupportPack** in our example):



For VMs (in this case, **Install VMware Tools**) set the condition as follows:

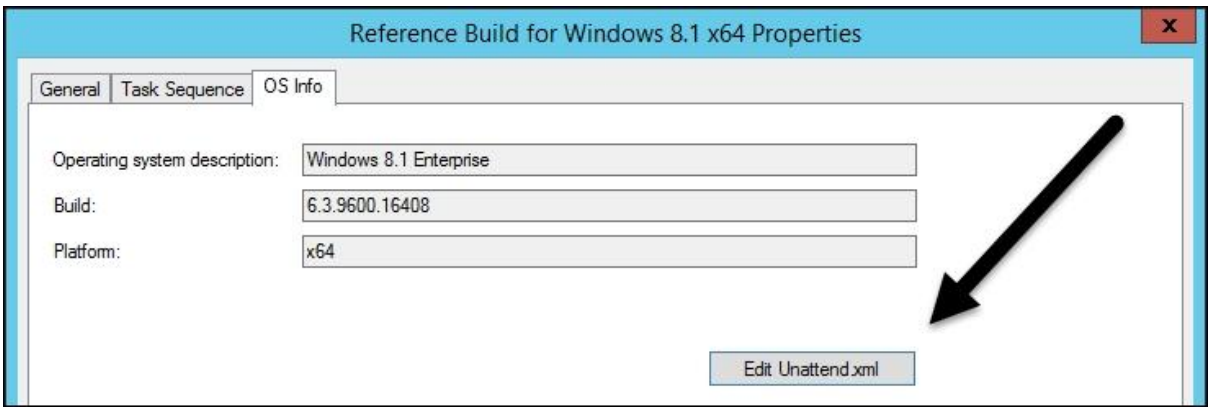


Using this method, we will force an application to run, but only in the conditions for which it is appropriate.

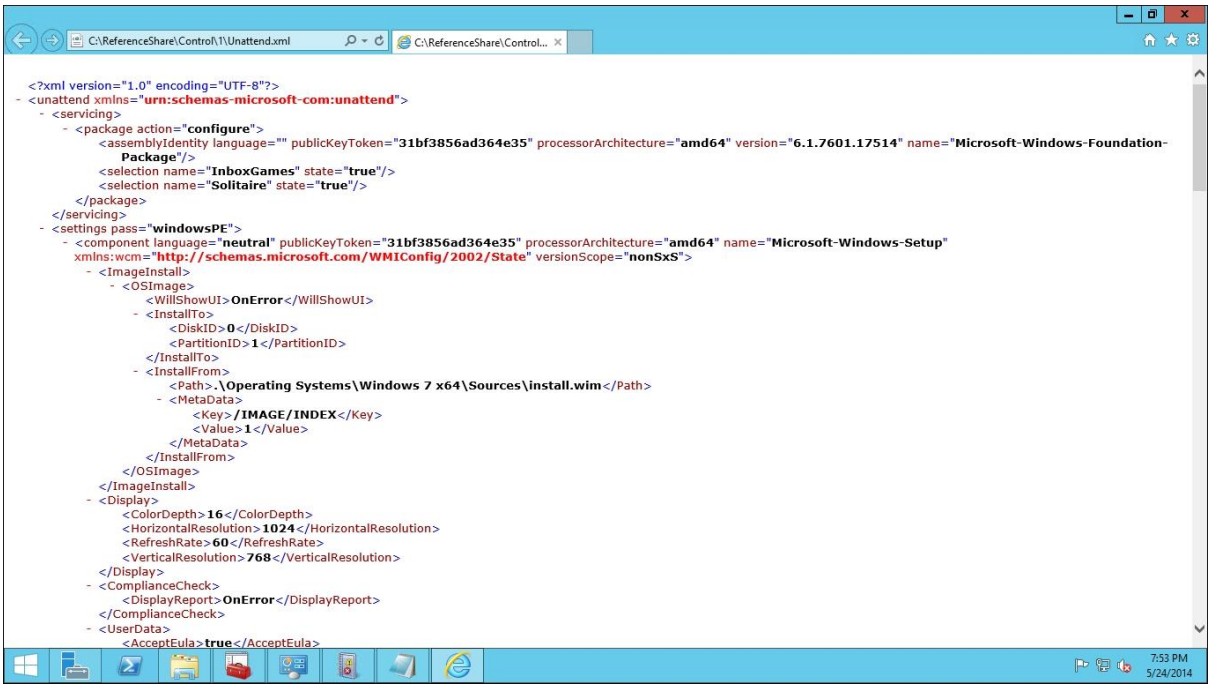
The Unattend.xml structure

Now that we have walked through the `CustomSettings.ini` structure, it is important to see exactly why this is relevant for an unattended installation of Windows. The key takeaway is that the variables, which MDT has created and defined, overlap in many cases with `Unattend.xml`. For example, we previously worked on a subsection in the `CustomSettings.ini` example that hinged on the `OSDComputerName` variable. In our `Unattend.xml` file, this is simply `<ComputerName>`.

We can review our `Unattend.xml` structure by going to our **Task Sequences** tab in our MDT console, right-clicking on a specific task sequence, selecting **Properties**, and then clicking on the **OS Info** tab, as shown in the following screenshot:



There is a **Edit Unattend.xml** button. This launches **Windows System Image Manager** (**WSIM**), so we won't see the raw file contents. To see this, we can easily browse to our reference share using the `C:\ReferenceShare\Control\1` path and the `Unattend.xml` file. Open the file, and by default, Internet Explorer will render it for you, as shown in the following screenshot:



However, to open this file for ease of modification, I prefer to use the keystrokes **Shift** + right-click and navigate to **Open with** | **Notepad**. It will then look like the following code snippet and is fully editable:

```
<?xml version="1.0" encoding="utf-8"?>
<unattend xmlns="urn:schemas-microsoft-com:unattend">
  <servicing>
    <package action="configure">
      <assemblyIdentity
name="Microsoft-Windows-Foundation-Package"
```

```
version="6.1.7601.17514" processorArchitecture="amd64"
publicKeyToken="31bf3856ad364e35" language="" />
      <selection name="InboxGames" state="true" />
      <selection name="Solitaire" state="true" />
    </package>
  </servicing>
  <settings pass="windowsPE">
    <component name="Microsoft-Windows-Setup"
processorArchitecture="amd64"
publicKeyToken="31bf3856ad364e35"       language="neutral"
versionScope="nonSxS"
xmlns:wcm="http://schemas.microsoft.com/WMIConfig/2002/State">
      <ImageInstall>
        <OSImage>
          <WillShowUI>OnError</WillShowUI>
          <InstallTo>
            <DiskID>0</DiskID>
            <PartitionID>1</PartitionID>
          </InstallTo>
          <InstallFrom>
            <Path>.\Operating Systems\Windows 7
x64\Sources\install.wim</Path>
            <MetaData>
              <Key>/IMAGE/INDEX</Key>
              <Value>1</Value>
            </MetaData>
          </InstallFrom>
        </OSImage>
      </ImageInstall>
```

TIP

It is not recommended to edit the `Unattend.xml` file manually, unless you really know what you are doing. The file is picky about the order, and some things that you think you might not need and want to delete are actually core setup process dependencies. So tinker at your own peril!

In the `Unattend.xml` file, we can see how MDT *bootstraps* itself as part of the logon process, as shown in the following code snippet:

```
<FirstLogonCommands>
  <SynchronousCommand wcm:action="add">
```

```
    <CommandLine>wscript.exe
%SystemDrive%\LTIBootstrap.vbs</CommandLine>
    <Description>Lite Touch new OS</Description>
    <Order>1</Order>
  </SynchronousCommand>
```

In the preceding code snippet, note that the first command run at logon is to launch `LTIBootstrap.vbs`, which then resumes and controls the MDT task sequence engine for the setup process.

The variables.dat structure

So, where do we get the list of the variables that exist? The Print-Ready Documentation for MDT 2013 is a perfect library, particularly the `Toolkit Reference.docx` file. To see the values that are being discovered for your hardware, one can easily open the `variables.dat` file. The structure is XML-based (one can simply rename the file to `.xml`), open it in Internet Explorer, and get a better view than Notepad.

However, when the `variables.dat` file is rendered in a browser, it appears as follows:

```
<MediaVarList Version="4.00.5345.0000">
  <var name="LOGPATH">
    <![CDATA[ X:\MININT\SMSOSD\OSDLOGS ]]>
    ...
    <![CDATA[ Bootstrap.ini ]]>
  </var>
  <var name="OSCURRENTVERSION">
    <![CDATA[ 6.3.9600 ]]>
  </var>
  <var name="OSCURRENTBUILD">
    <![CDATA[ 9600 ]]>
  </var>
  <var name="OSVERSION">
    <![CDATA[ WinPE ]]>
  </var>
  <var name="ISSERVEROS">
    <![CDATA[ False ]]>
  </var>
  <var name="ISSERVERCOREOS">
    <![CDATA[ False ]]>
  </var>
```

```
...
<var name="HOSTNAME">
  <![CDATA[ MININT-I7GS8HP ]]>
</var>
<var name="ASSETTAG">
  <![CDATA[ 7774-6450-3382-1242-9318-1886-08 ]]>
</var>
<var name="SERIALNUMBER">
  <![CDATA[ 7774-6450-3382-1242-9318-1886-08 ]]>
</var>
<var name="MAKE">
  <![CDATA[ Microsoft Corporation ]]>
</var>
<var name="MODEL">
  <![CDATA[ Virtual Machine ]]>
</var>
<var name="PRODUCT">
  <![CDATA[ Virtual Machine ]]>
</var>
<var name="UUID">
  <![CDATA[ 41498066-DE4A-4C40-918F-F7C8F8BD32D6 ]]>
</var>
<var name="MEMORY">
  <![CDATA[ 2047 ]]>
</var>
<var name="ARCHITECTURE">
  <![CDATA[ X64 ]]>
</var>
<var name="PROCESSORSPEED">
  <![CDATA[ 3400 ]]>
</var>
<var name="CAPABLEARCHITECTURE">
  <![CDATA[ AMD64 X64 X86 ]]>
</var>
<var name="ISLAPTOP">
  <![CDATA[ False ]]>
</var>
<var name="ISDESKTOP">
  <![CDATA[ True ]]>
</var>
```

So, here we have our variables that we called previously, `ISDESKTOP` and `ISLAPTOP`.

CustomSettings.ini and the Unattend.xml file

Now that we've reviewed the `CustomSettings.ini` structure, some commonalities between the `Unattend.xml` file and `CustomSettings.ini` may be evident. For example, the `OSDComputerName` value of `CustomSettings.ini` is what will dynamically supersede our task sequence's `<ComputerName>` value of `Unattend.xml` when the task sequence runs.

This is the key value in using MDT for deployment. With a structured `Unattend.xml` file and deployment, you get a hands-off deployment. However, it's machine-specific, typically, hardware-specific. It's also a challenge to manage this and change `Unattend.xml`, even with WSIM. So, this solution of utilizing `Unattend.xml` manually (even per model, for example, for driver considerations) doesn't scale and should not be used in the Enterprise environment, or small and medium businesses.

Dynamic modification

With MDT, we are essentially scaling `Unattend.xml` and dynamically modifying the values as we see fit, for each task sequence run. This is the value of utilizing this system for deployments. If you need to take it even further, you can dynamically modify `CustomSettings.ini` in the manner that we saw earlier in this chapter. However, this modification method of using the scripting environment, native to the MDT, doesn't scale very well.

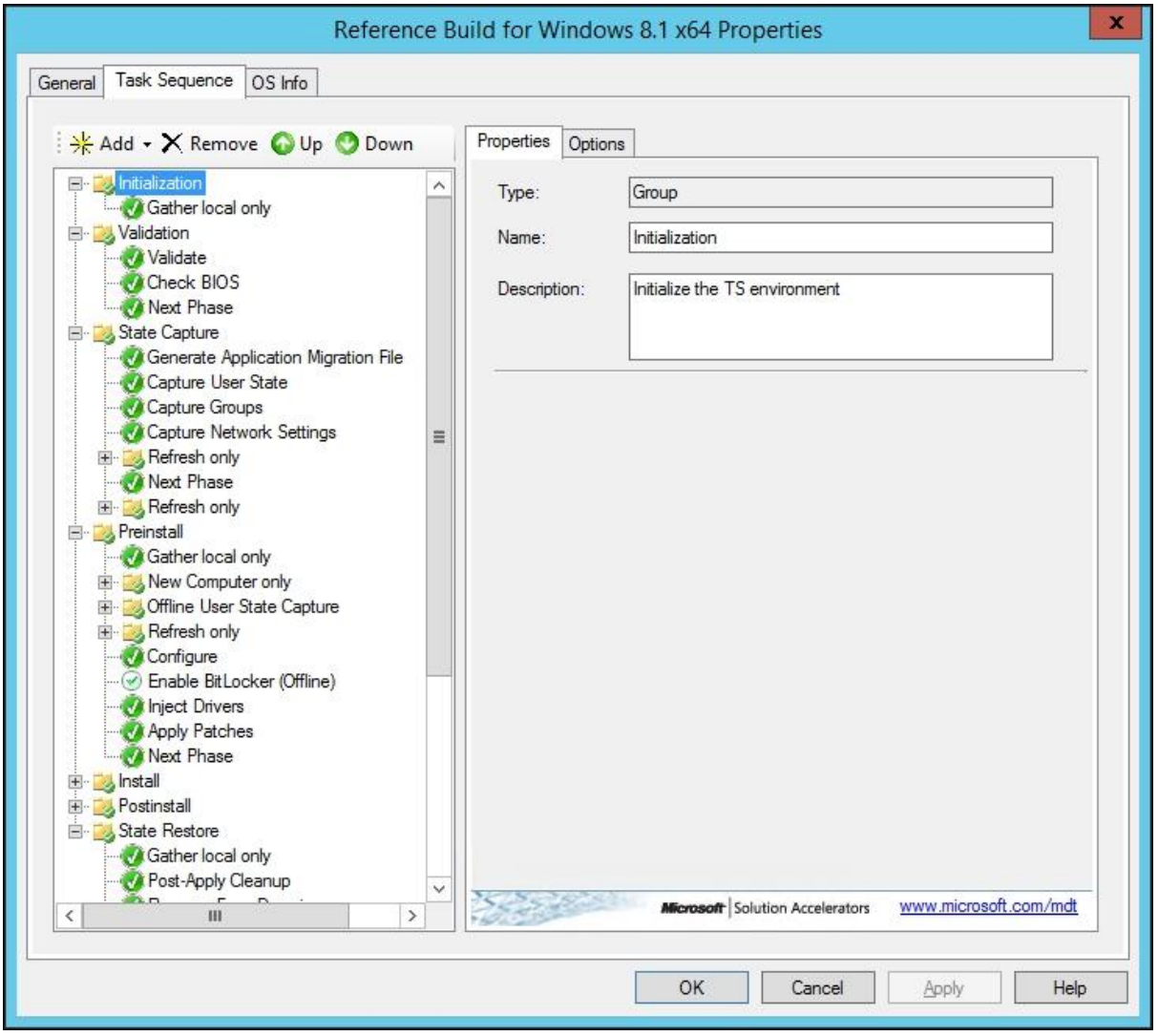
The next option would be to utilize a concept known as **UserExit scripts** to modify the `CustomSettings.ini` dynamically. Essentially, UserExit scripts are a way to exit the MDT scripting environment and perform some function call and come back and populate variables based on the function call. An example of this might be https://blogs.technet.microsoft.com/deploymentguys/tag/user-exit-script/, which details the setting of the `TaskSequenceID` variable in `CustomSettings.ini` based on the available RAM installed.

Lastly, there is another concept to dynamically modify `CustomSettings.ini`, and that is the MDT SQL DB option. In this setup, a Microsoft SQL Database (installed on the MDT Server or an instance of another existing Microsoft SQL Server in the environment) services the deployment share and `CustomSettings.ini`. In this configuration, you can dynamically populate groups of applications based on role, manage complex selection profile arrangements, or even pre-populate the database

with MAC addresses and automatically pick `TaskSequenceID` based on the MAC address of the machine.

Task sequence structure

If we examine the structure of our task sequence, we can see it is broken into a set of folders and underlying actions for each folder (and in some cases, a tree of folders under the top-level folder), as shown in the following screenshot:



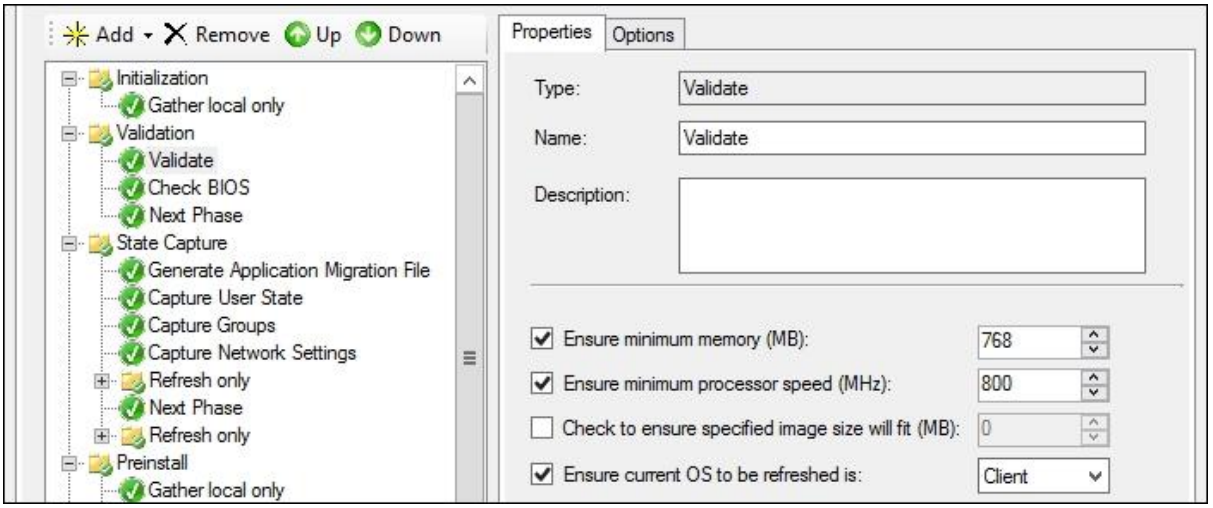
The root-level steps and their functions are explained in the following sections.

Initialization

In this step, the task sequence is initialized and a **Gather local only** step is run. This gathers the data utilizing the script, `ZTIGather.wsf`, and in this step, the original `variables.dat` file is populated.

Validation

In this step, the task sequence checks the local environment to verify that it is suitable to run the rest of the task sequence. In this step, for instance, we will verify that the system has a minimum of `768` MB to run Windows PE, and the processor speed is at least `800` Mhz:



TIP

Microsoft Virtual PC had an issue where it would report 0 Mhz for processor speed, so when you ran a task sequence in Windows 7 Virtual PC, you had to uncheck this step or the task sequence would fail.

The next part of this step is an interesting component, **Check BIOS**. This is placed solely for the purpose of validating BIOS. So if you have a model of computer that requires a particular version of BIOS to allow the operating system to install properly, you can modify the file in `\Scripts\ZTIBIOSCheck.xml`, which `ZTIBIOSCheck.wsf` calls, and this guidance is provided in the XML file itself:

NOTE

If you have a computer BIOS version that needs to be identified during the BDD installation process, you can run the following VBScript program (`ZTINextPhase.wsf`) to extract out the necessary fields on the computer system.

Then, we will simply run the step, **Next Phase**, which calls `ZTINextPhase.wsf`, writes to the `variables.dat` file (for future reference) and logs (for troubleshooting), and then continues to the next step.

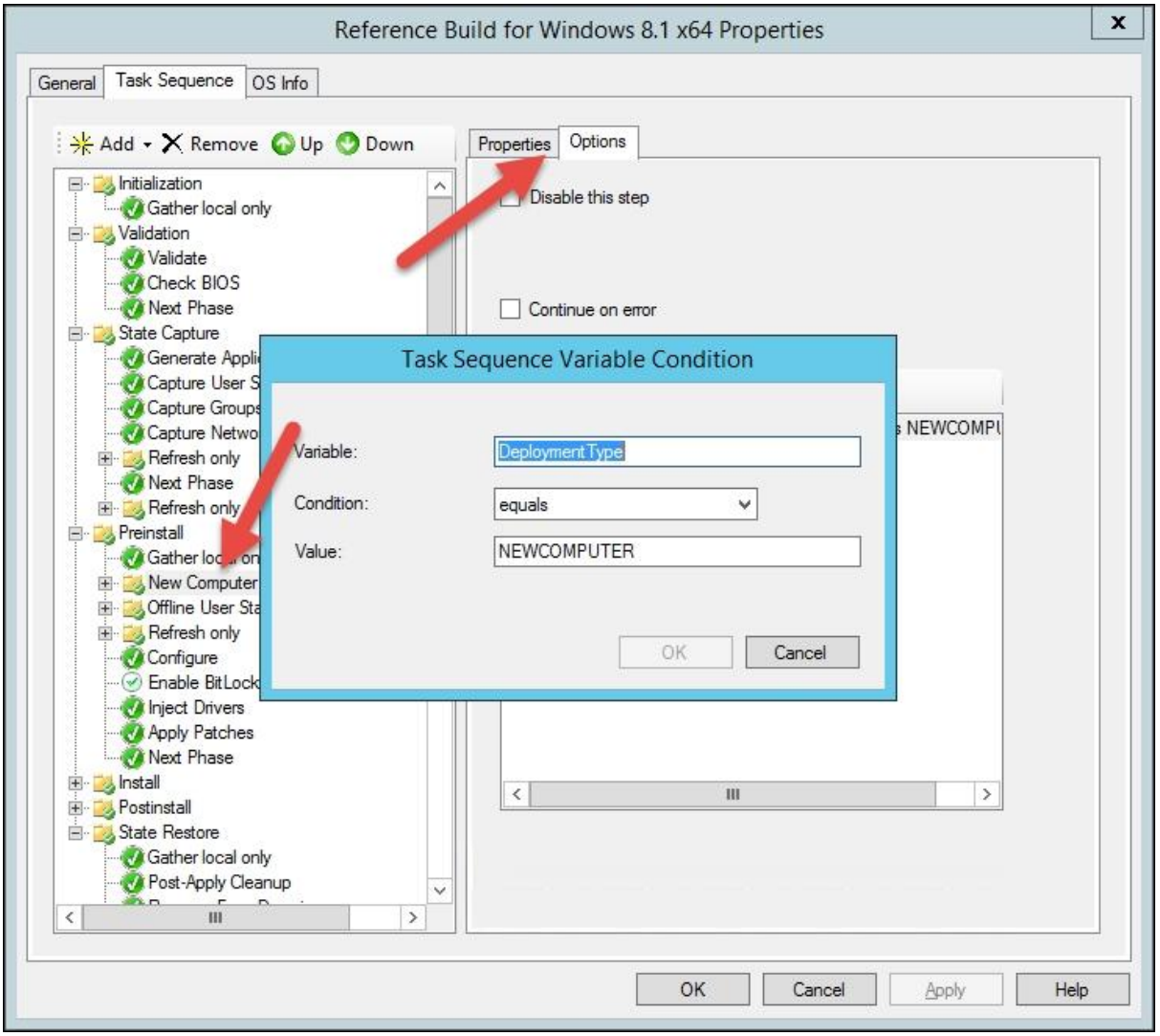
State capture

In this step, the task sequence may create the application migration file, capture user state by calling the **User State Migration Tool** (**USMT**), appropriate XML

configuration files, capture network settings, and so on. I use the word *may* as it will only run these steps if they are called by the task sequence that is currently executing. This is also the area where the **Refresh** steps of a task sequence also may be run, particularly the **Disable BDE Protectors** (to disable BitLocker drive encryption, or suspend it temporarily) and **Apply Windows PE** steps.

Preinstall

Firstly, the **Preinstall** task sequence step runs another **Gather local only** step. Then, it gets interesting. If the task sequence is running in a situation that meets the **NEWCOMPUTER** definition (as defined by the **Options** tab of the **Task Sequence** folder), then it runs the steps under this folder, as shown in the following screenshot:



If you observe, under this folder, if it runs, the task sequence will format and partition the hard drive and then copy scripts to be run locally and set up a logging area. If it doesn't run, it will skip to **Offline User State Capture**, or **Refresh only**, again based on the scenario.

After the variables are played out in the task sequence, an **Inject Drivers** occurs. This is an interesting step, where according to the selection profile specified, drivers are injected for the operating system. The `ztidrivers.wsf` script is called, which is described in detail in the word document, `Toolkit Reference.docx`, from the MDT 2013 Documentation. Basically, what happens is a query is made for **Plug and Play** (**PnP**) IDs located on the hardware on which the script is running, and then a query is made against the XML file that is the master inventory of the deployment share's `Out-of-Box Drivers` directory (where you would import the third-party drivers) and any matches are copied over and made available for installation.

Now, one key concept here is, as I mentioned, according to the selection profile specified. So this step, by default, would pick *every* driver as a possibility for the installation of Windows. In some instances, it is okay to leave this as a default, let the PnP magic happen in the scripts, and auto-pick your drivers.

What if you have two Lenovo laptops, both of which resolve a piece of hardware that they have installed to the same PnP ID, but there are actually two drivers for this PnP ID in your `Out-of-Box Drivers` folder? What happens? Which driver is installed on which model? It's somewhat random. Therefore, in this scenario, for example, the mass storage device (your SATA controller, for instance, that connects the system board to the SSD/HD of the device) might have the correct driver and allows the system to boot fine, or if the wrong driver were to be loaded, the system would blue screen, probably quite quickly, with **STOP 0x0000007B (INACCESIBLE_BOOT_DEVICE)**, something such as documented at http://support.microsoft.com/kb/324103 , for instance.

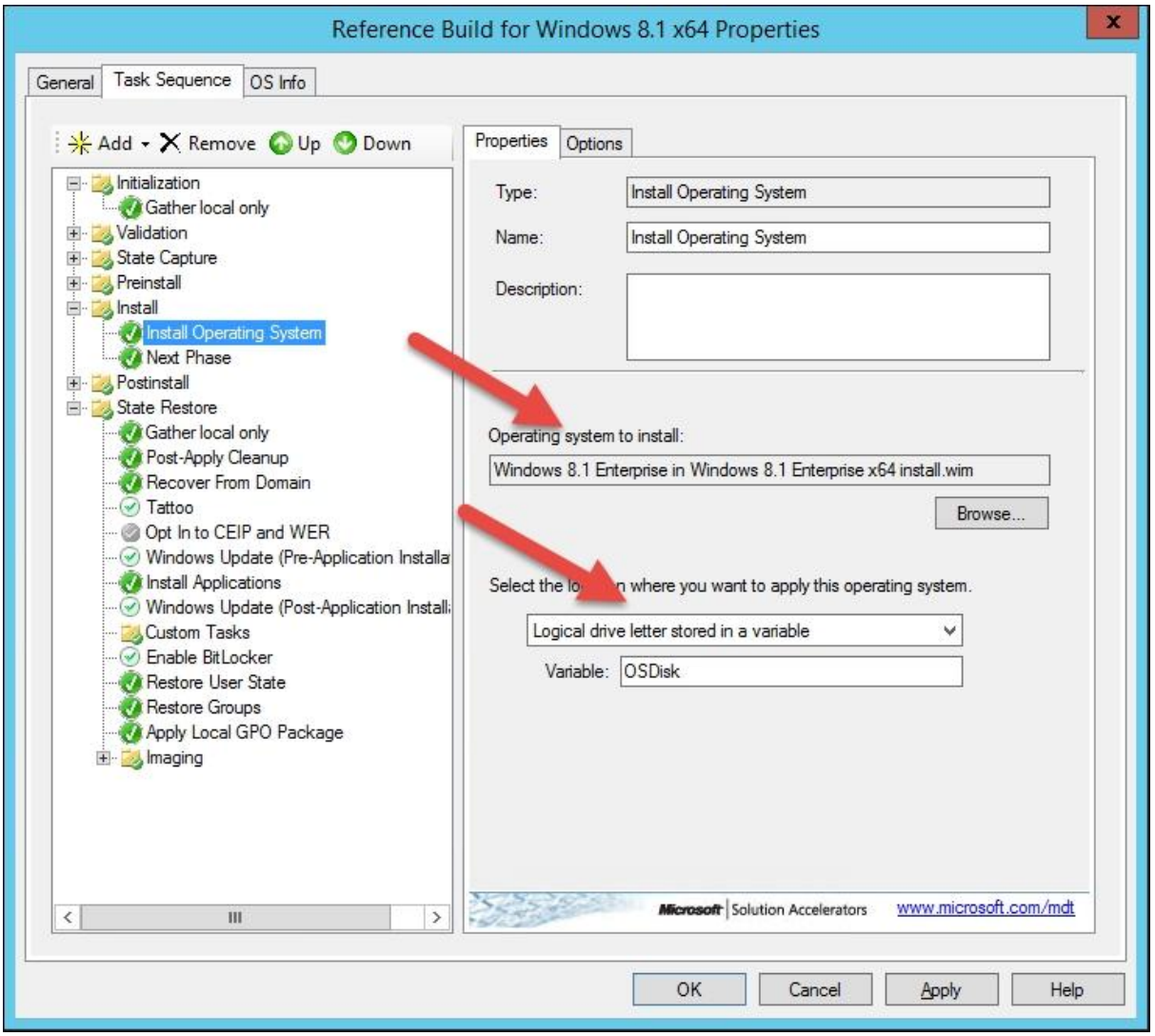
For this and other considerations, refer to Chapter 6 , *Drivers*. The key to note here is that you can specify alternate selection profiles to address this potential hiccup in your deployment.

After the **Inject Drivers** step is complete, **Apply Patches** runs. This is similar to the **Inject Drivers** step, but instead it utilizes DISM to apply Windows Updates, Language Packs, and so forth to the WIM offline. It retrieves these from the `Packages` directory of the deployment share. Later on, we'll discuss a step to download updates from Windows Updates, or a **Windows Server Update Services** (**WSUS**) Server, for online action. However, for now, we offline-install patches in this step.

Install

In the install phase of task sequence, the actual operating system is applied. What is actually happening is a script is running, which calls DISM/Apply-Image to apply the WIM that you select onto the storage you select. This is modifiable by the way!

Note that in the following screenshot, the WIM has a **Browse...** button to select an alternate WIM, for instance, if you wanted to switch from Windows 8.1 with Update 1 to Windows 8.1 with Update 2 if and when it comes out. You can also select the logical drive to be applied to the OS being installed:



Postinstall

In the **Postinstall** task sequence phase, scripts are copied to the drive that we previously formatted and performed the DISM /Apply-Image to, and then driver injection occurs and a **Windows Recovery Environment** (**WinRE**) is applied to the volume. Then, the machine is rebooted. It is pretty simple and straightforward.

State restore

In the **State Restore** phase, a domain join action—**Recover From Domain—**can run to join the machine to the domain automatically. The registry is tattooed, options to opt in to CEIP and WER are presented, and then Windows Updates can run (though it is disabled by default). This will hit the Internet Windows Updates servers,

unless one specifies a value in `CustomSettings.ini` for WSUS server Then, the step will point directly to the specified WSUS server, which ordinarily should be a quicker and less bandwidth-hungry operation.

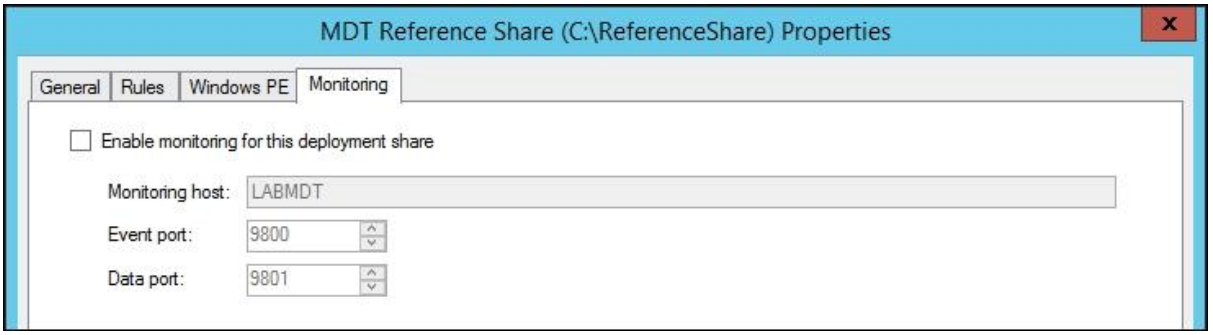
TIP

This WSUS server *cannot* be an SCCM server or WSUS server touched by SCCM. It is not supported and odd update experiences will occur.

Logging

In MDT 2010, an option to log the task sequence runs was introduced as a variable. This `SLShareDynamicLogging` variable requires you to point to a UNC path where deployments would be written to in real time. Therefore, as Michael Niehaus pointed out in his blog back in 2009, you could run `Trace32.exe` against the `BDD.LOG` file in the share for each deployment and see it happening in real time versus waiting for the deployment to fully complete and then harvesting `BDD.LOG` from the deployed or failed deployment machine.

In MDT 2012, a feature known as MDT Monitoring was introduced. You can find the settings for it by right-clicking on the deployment share and selecting **Properties**. The last tab is **Monitoring** and should look similar to the following screenshot, by default:



To enable monitoring, simply tick the box and click on **Apply**. Note that checking the box here adds a line to our `CustomSettings.ini` and rules entry, `EventService=http://LABMDT:9800`. Also the default share used is a directory on the `C:\events` local MDT host. This is a configurable option using the MDT task sequence variable, `EventShare`, where `EventShare=\\host\events`. Also, a service must be running for this to work, it is installed automatically and enabled when you click **Apply**.

TIP

One can also utilize PowerShell to access the data in the monitoring service using the cmdlet `Get-MDTMonitorData`; an example is as simple as `(Get-MDTMonitorData -Path DS001:).Count`

NOTE

Windows 10

Windows 10 does not significantly modify the instructions of this chapter. Task sequence customization continues as any other operating system at the time of writing this.