Addendum 1: Windows 10 IoT Enterprise Build 10240

By Sean D. Liming and John R. Malin
Annabooks – www.annabooks.com

November 16, 2015

OS: Windows 10 IoT Enterprise (a.k.a Windows 10 Enterprise LTSB)
Build Number: 10240

1.1 Introduction: Book Addendum

Now that Windows 10 has been released, we can confirm that the development process for embedded OEMs will not really change much. If you are going from Windows Embedded 8.1 Industry to Windows 10, the development process with System Image Manager is the same, thus this paper serves as an addendum to the book: *Starter Guide for Windows System Image Manager*. The addendum will discuss the product naming, changes to activation, new tools coming, changes to the Lockdown Features, code examples for customizing programmatic management of the Lockdown Features.

1.2 What's in a Name

The Windows 10 launch is one of the most confusing launches we have seen. Microsoft wants to focus on the cloud rather than the operating system, but devices need to run an OS to run apps that exchange data with the cloud. Microsoft didn't have a web page updated to discuss the new Windows 10 IoT product line. We wrote the article *Windows 10 IoT – The Big Reboot* to help explain what has been launched. Only recently has something appeared from Microsoft - http://www.microsoft.com/en-us/WindowsForBusiness/windows-iot.

Looking at the three products Windows 10 IoT Enterprise is the latest release of the desktop line for Embedded/IoT. Windows 10 IoT Enterprise continues the road map:

- Windows NT Embedded (NTe)
- Windows XP Embedded (XPe)
- Windows Embedded Standard 7 (WES7)
- Windows Embedded 8 Standard (WE8S)
- Windows Embedded Industry 8.1 (WEI8.1)
- Windows 10 IoT Enterprise

Now, we are going to have some fun with names. If you look for a product called Windows 10 IoT Enterprise, you will not find it. There are two Windows 10 Enterprise versions:

1. Windows 10 Enterprise
2. Windows 10 Enterprise LTSB

We know there are subversions to address different international compliances, but these are the main two. The LTSB (Long Term Service Branch) version is the one Embedded/IoT device OEMs will use to build their systems. Windows 10 IoT Enterprise is really Windows 10 Enterprise LTSB.

If the acronym LTSB sounds familiar, it is because Linux distributions use LTSB to manage long term distribution. Microsoft is using LTSB for embedded developers to lock in an OS and support updates for a set number of years. The non-LTSB version will continue to get feature updates. In some future date, a new LTSB will be released with the latest features.

LTSB approaches breaks with the traditional shared success model: where you can test your application on Windows desktop and then bring the application to Windows Embedded. There were some breakdowns and quirks with Windows XP Embedded, but later versions matched up

Copyright © 2015 Annabooks, LLC. All rights reserved

well. If the latest feature like IE or Media Player was missing, you could download the update separately. With the new LTSB approach, you would have to go back to an old release of Windows that might not have the features your current Windows desktop has. Here is a real example: Microsoft forgot to include the Keyboard Filter feature with build 10240. The Keyboard Filter could be made available in the next non-LTSB release, but it would not be available for Embedded/IoT developers for a few years until the next LTSB release.

There is some benefit to this approach. As new features become available, they will be well tested in the non-LTSB releases before being put into a LTSB release.

1.3 Activation Story

Speaking of things that don't make sense, the activation story has changed. After activation was forced on embedded/IoT developers in WE8S and WEI8.1, Windows 10 IoT Enterprise removes the requirement. WE8S and WEI8.1 were non-starts for embedded/IoT OEMs. The lack of sales, certain product requirements such as CRC checks, and informative customer feedback, convinced Microsoft to change direction on activation.

So long as your system doesn't connect to the Internet, activation is not required. As soon as a connection to the internet is made, activation will take place.

1.4 Development Tools

The development process using System Image Manger (SIM) to create a custom answer file and configuration set is the same. You can download the latest ADK for Windows 10 here:

https://msdn.microsoft.com/en-us/windows/hardware/dn913721%28v=vs.8.5%29.aspx?f=255&MSPPError=-2147217396

Our book *Starter Guide for Windows System Image Manager* is still relevant with a few changes, which we will cover in the next session.

There is a new tool in the ADK called Windows Imaging and Configuration Designer (Windows ICD), which Microsoft is promoting as the custom OS build tool of the future. Windows ICD works great with Windows 10 IoT Core, but we found Windows ICD didn't work well with Windows 10 IoT Enterprise. After our testing, we feel Windows ICD doesn't meet the needs of embedded developers. Presetting the lockdown features didn't work, and syspreping the image was problematic. WINDOWS ICD shows promise, but we will have to wait for a few releases so the unintended features get worked out. For now we recommend sticking with SIM.

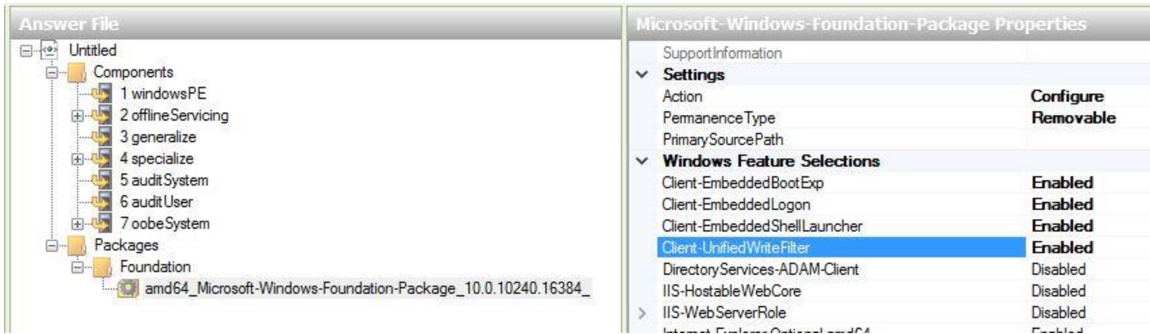
1.5 Lockdown Feature Changes

The biggest change between WEI 8.1 and Windows 10 IoT Enterprise is the Lockdown Features. We were present when it was stated that Windows 10 IoT Enterprise would have the same or similar lockdown features as WEI 8.1. Something must have change along the way. Keyboard Filter not available was mentioned already. The other features not available in Windows 10 IoT Enterprise are Embedded Lockdown Manager, App Launcher, USB Filter, and Gesture Filter.

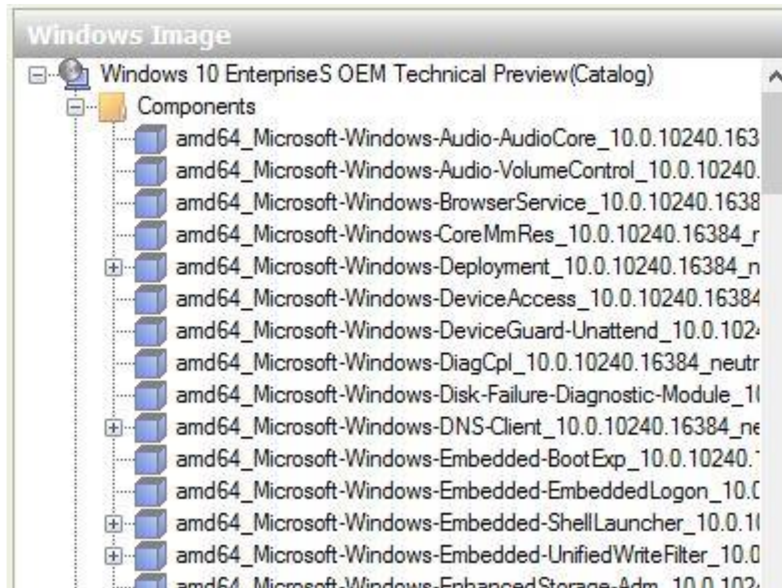
Unified Write Filter, Shell Launcher, Embedded Boot Experience, and Embedded logon are the lockdown features that are currently available in build 10240.

The features are enabled in the Windows Foundation Package. The ISKU prefix has been replaced with the Client, thus making these features appear a more natural fit with Windows.

Copyright © 2015 Annabooks, LLC. All rights reserved



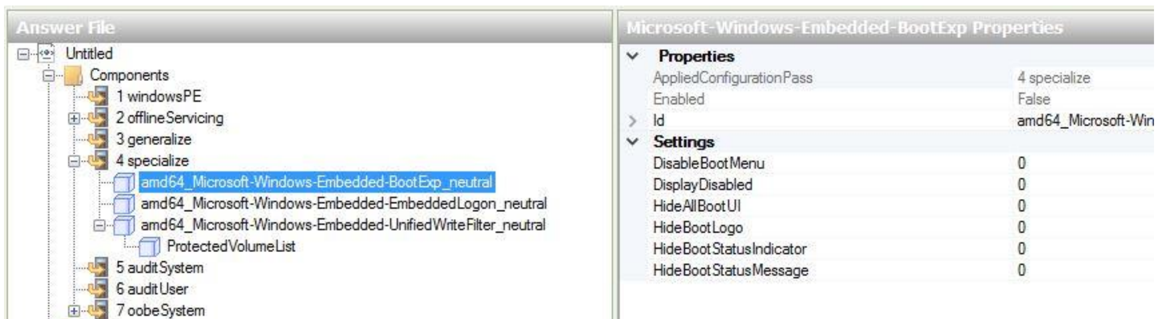
The components to preset the settings have not changed.



The component settings for each feature have not changed from WEI8.1. For completeness, the following sections provide the details.

1.5.1 Embedded Boot Experience

The Embedded Boot Experience enables and disables items that are seen on boot. Again, a custom logo is not supported on boot up unless you have a system with UEFI firmware and use the BGRT image.



Here are the available settings:

Copyright © 2015 Annabooks, LLC. All rights reserved

- **DisableBootMenu** – an integer value that disables F8 and F10 during boot-up for gaining access to advanced boot options. Possible values are 1 for disable the menu, and 0 to enable the menu.
- **DisplayDisabled** – an integer value that controls whether a blank screen appears when an error is encountered. Possible values are 1 to display a blank screen or 0 to show the error.
- **HideAllBootUI** – an integer value when set to 1 will suppress all Windows UI elements on startup – logo, status indicator, and status message. A value of 0 will not enable all, but you can suppress individual elements with the following settings.
- **HideBootLogo** – an integer value that can suppress the Windows logo from appearing on startup. Possible values are 1 to suppress the logo, and 0 to not suppress.
- **HideBootStatusIndicator** - an integer value that can suppress the status indicator on startup. Possible values are 1 to suppress the status indicator, or 0 to not suppress.
- **HideBootStatusMessage** – an integer value that can suppress status messages from appearing on startup. Possible values are 1 to suppress the status messages, or 0 to not suppress.

A blank screen on boot up can be a concern to some users. To at least show something on startup and not tip off the users that the OS is Windows, we typically set DisableBootMenu, HideBootLogo, and HideBootStatus Message to 1.

1.5.2 Embedded Logon

Microsoft-Windows-Embedded-EmbeddedLogon component settings suppress Windows UI elements that are displayed during the logon and shutdown processes.

Here are the available settings:

- **AnimationDisabled** - an integer value that enables or disables logon screen transition animation. The possible values are 1 to disable and 0 to enable.
- **BrandingNeutral** – an integer value that specifies what logon UI elements appear. There are several values that can be combined using bitwise OR logic.

  1 - Disables all logon screen UI elements.
  2 - Disables the Shutdown button.
  4 - Disables the Language button.
  8 - Disables the Ease of access button.
  16 - Disables the Switch user button.
  32 - Disables the Blocked Shutdown Resolver (BSDR) screen.

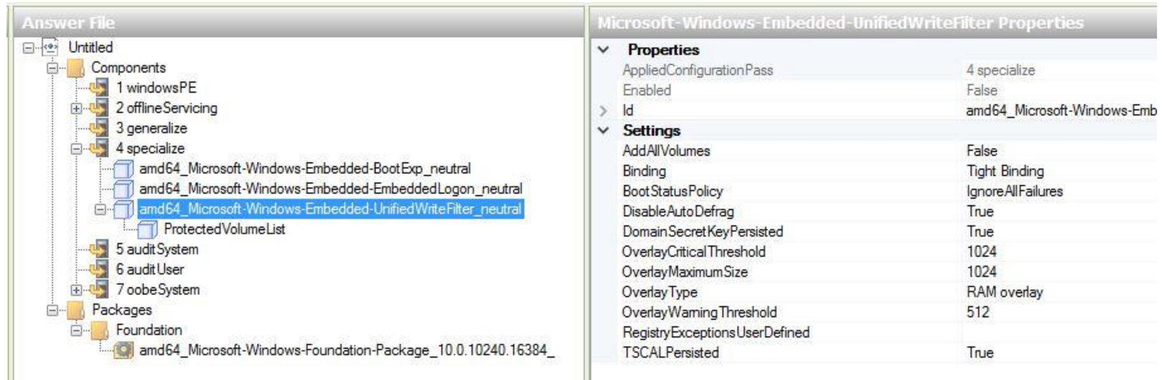
- **HideAutoLogonUI** – an integer value that either shows or hides logon screens when automatic logon is enabled. Possible values are 1 to hide logon screens, or 0 to show logon screens.

Copyright © 2015 Annabooks, LLC. All rights reserved

- **NoLockScreen** – an integer value that either enables or disables the Lock screen and UI elements. Possible values are 1 to disable lock screen, or 0 to enable lock screen.
- **UIVerbosityLevel** – a hexadecimal value that either enables or disables status messages during startup, logon, and shutdown. Possible values 0x1 to disable status messages, or 0x4 to enable status messages.

1.5.3 Unified Write Filter

The Microsoft-Windows-Embedded-UnifiedWriteFilter component has a combination of settings to protect disk and registry. The following figure shows the different settings that can be preset in the answer file.



UWF settings can be broken down into 3 different groups. The first group is the generic settings:

- **AddAllVolume** – True or False – Specifies whether all volumes on the device are write-protected. The default is false. This could be used to prevent writes to removable devices if they are attached.
- **Binding** – Provides two options: 1. Loose Binding to enable specifying a volume by using a drive letter. 2. Tight Binding to specify a volume by using the volume device identifier. Tight Binding is the default.
- **BootStatus Policy** – Allows you to display different boot failures. The default is to IgnoreAllFailures.
- **DisableAutoDefrag** – True or False – The default is set to True so automatic disk defragmentation will not run.

The second group is for the Registry Filter:

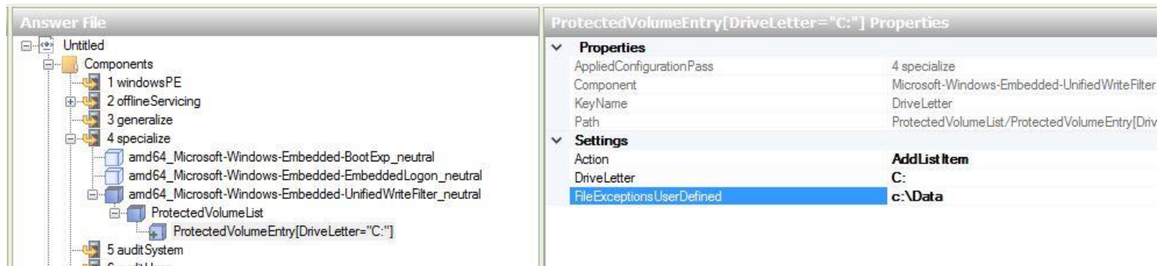
- **DomainSecretKeyPersisted** – True or False – The default is True, so the Domain Secret Key will be preserved between reboots.
- **TSCALPersisted** – True or False – The default is True, so the TSCAL is preserved between reboots.
- **RegistryExceptionsUserDefined** – These are the registry keys that you define to be preserved between reboots. Only HKLM keys can be preserved. HKCU keys are not supported.

The third group is for the write-filter:

- **OverlayWarningThreshold** – Integer value in megabytes (MB) that specifies the overlay warning threshold and sends out a warning event notification when UWF overlay size has reached or exceeded this value.
- **OverlayCriticalThreshold** - Integer value in megabytes (MB) that specifies the overlay critical threshold and sends out a critical threshold notification event when UWF overlay size has reached or exceeded this value.

Copyright © 2015 Annabooks, LLC. All rights reserved

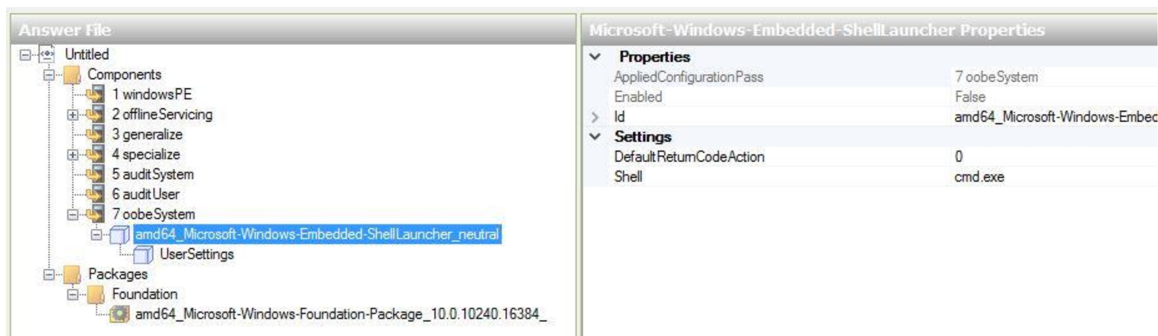
- **OverlayMaximumSize** – Integer value in megabytes (MB) that specifies maximum overlay size. The default setting 1024 MB
- **OverlayType** – There are two options: RAM overlay and Disk Overlay. The default setting is RAM.
- **ProtectedVolumeList** – Allows you to define the volumes to be protected by UWF and any write-through files and folders. You must right-click on the sub-component to add a new protected volume and any associated write-through sections. The write-through list must contain absolute paths (i.e. c:\data). The following Figure shows an example.



1.5.4 Shell Launcher

Shell Launcher allows you to launch a tradition windows application as the shell. Any tradition application can be the shell to the system. Shell Launcher also allows you to set a custom shell per user.

Note: The PowerShell script and or WMI call is still needed to enable Shell Launcher after OS installation.



Here are the settings:

- **Shell** – Points to the application to be launched as the default shell. You can either set the path or call the application by name, if it is in the execution path already. If this is a custom application, you will want to create a module for it. A dependency on Shell Launcher is not a must.
- **DefaultReturnCodeAction** – An integer value that specifies the action to take when the default application exits. Possible Values are 0 – restart shell, 1 – reboot, 2 – shut down, or 3 – do nothing.
- **UserSettings** – Allows you to specify the shell and shell exit action for a specific user account. This shell will be launched over the default shell. You can have different applications launch as the shell for different users.

1.6 WMI Calls Replace the Missing Embedded Lockdown Manager

The lack of Embedded Lockdown Manager makes post OS installation management a little more fun. WMI support for UWF and Shell Launcher were introduced in WE8S, and are now the

Copyright © 2015 Annabooks, LLC. All rights reserved

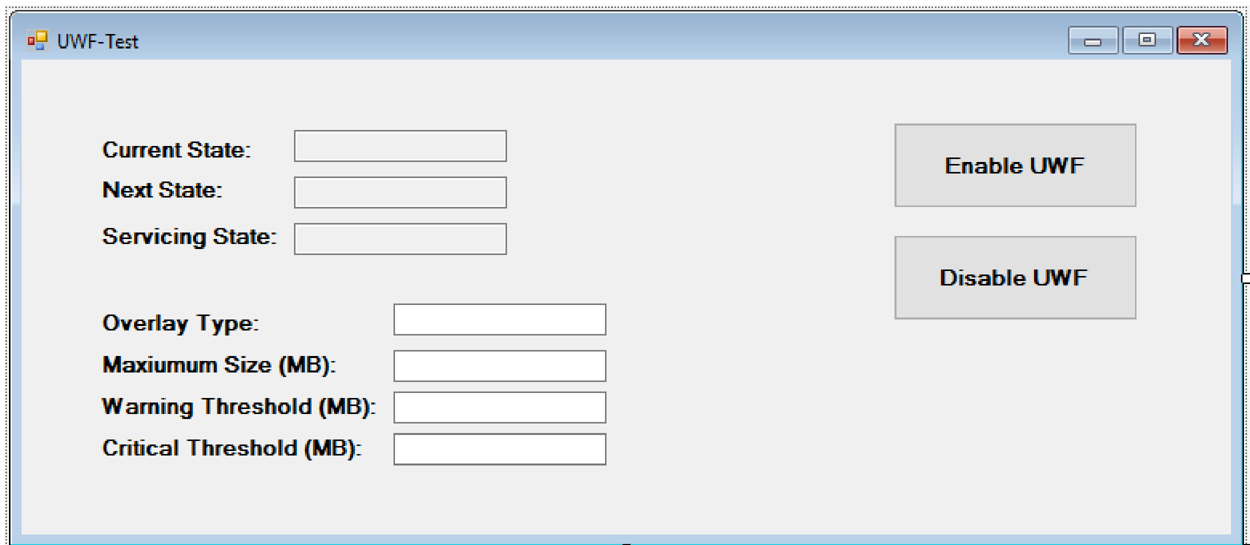
primary management solution for Windows 10 IoT Enterprise until something else is introduced. Like the features themselves, not much has change from WEI 8.1, thus the WEI 8.1 references are still valid:

UWF WMI reference can be found here: https://msdn.microsoft.com/en-us/library/dn449427(v=winembedded.82).aspx

Shell Launcher WMI reference can be found here: https://msdn.microsoft.com/en-us/library/dn449372(v=winembedded.82).aspx

1.6.1 UWF Example

UWFMGR.EXE is still available. Many developers perform a process start call to the Uwfmgr.exe to perform basic operation such as enabled and disable. WMI classes can be integrated into a custom application. Here are some C# code snippets based on a tradition Windows form application shown below.



Enabled UWF

```
ManagementScope scope = new ManagementScope(@"root\standardcimv2\embedded");
ManagementClass UWFFilter = new ManagementClass(scope.Path.Path, "UWF_Filter",
null);

foreach (ManagementObject mo in UWFFilter.GetInstances())
{
   ManagementBaseObject UWFEnable = mo.InvokeMethod("Enable", null, null);
}
```

Disable UWF

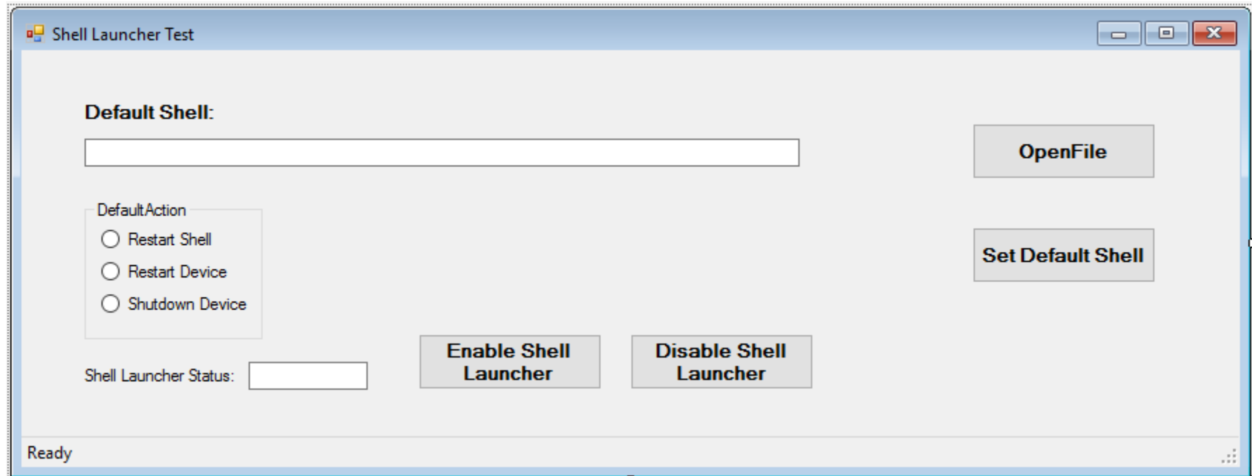
```
ManagementScope scope = new ManagementScope(@"root\standardcimv2\embedded");
ManagementClass UWFFilter = new ManagementClass(scope.Path.Path, "UWF_Filter",
null);

foreach (ManagementObject mo in UWFFilter.GetInstances())
{
   ManagementBaseObject UWFEnable = mo.InvokeMethod("Disable", null, null);
```

Copyright © 2015 Annabooks, LLC. All rights reserved

```
}
```

1.6.2 Shell Launcher Example

There is no command line utility for Shell Launcher so WMI is the only way to manage the Shell Launcher post OS install. Here are some C# code snippets for Shell Launcher based on a tradition Windows form application shown below



Enable Shell Launcher:

```
ManagementScope scope = new ManagementScope(@"root\standardcimv2\embedded");
ManagementClass WESL = new ManagementClass(scope.Path.Path, "WESL_UserSetting",
null);
ManagementBaseObject Enabledparams = WESL.GetMethodParameters("SetEnabled");

Enabledparams["Enabled"] = true;
WESL.InvokeMethod("SetEnabled", Enabledparams, null);
```

Disable Shell Launcher:

```
ManagementScope scope = new ManagementScope(@"root\standardcimv2\embedded");
ManagementClass WESL = new ManagementClass(scope.Path.Path, "WESL_UserSetting",
null);
ManagementBaseObject Enabledparams = WESL.GetMethodParameters("SetEnabled");

Enabledparams["Enabled"] = false;
WESL.InvokeMethod("SetEnabled", Enabledparams, null);
```

Set Default Shell:

```
ManagementScope scope = new ManagementScope(@"root\standardcimv2\embedded");
ManagementClass WESL = new ManagementClass(scope.Path.Path, "WESL_UserSetting",
null);
ManagementBaseObject SetDSParams = WESL.GetMethodParameters("SetDefaultShell");

SetDSParams["Shell"] = txtDefaultShell.Text;
if (rbRestartApp.Checked == true)
{
```

Copyright © 2015 Annabooks, LLC. All rights reserved

```
   SetDSParams["DefaultAction"] = 0;
}
if (rbRestartDevice.Checked == true)
{
   SetDSParams["DefaultAction"] = 1;
}
if (rbShutdownDevice.Checked == true)
{
  SetDSParams["DefaultAction"] = 2;
}

WESL.InvokeMethod("SetDefaultShell", SetDSParams, null);
```

Copyright © 2015 Annabooks, LLC. All rights reserved