Writing User Interfaces with WPK

Quick Start

The Windows 7 Resource Kit companion CD includes many Windows PowerShell modules that you can use, but you might want to pay special attention to the WPK module. WPK or the "Windows Presentation Foundation (WPF) PowerShell Kit," it enables you to use Windows PowerShell commands to create and display rich user interfaces. With WPK, systems administrators and beginning scripters can make their jobs easier by building user interfaces to simplify complex tasks.

This Quick Start topic will show how to use WPK to create the following real-world interfaces.

* A dialog box that collects information from the user
* A simple process monitor
* A media player

To start,

1. Copy the WPK folder to your Modules directory.

$home\Documents\WindowsPowerShell\Modules

For example:

C:\Users\<User-Name>\Documents\WindowsPowerShell\Modules

1. Start Windows PowerShell ISE (Integrated Scripting Environment).
2. At the Windows Powershell ISE command prompt, type:

import-module WPK

The first time that you import the WPK module, it takes approximately one minute to generate the code that it needs to manage the Windows Presentation Foundation.

The functions in the WPK module generate Windows PowerShell script code, so you can use them to create scripts as you discover more and more useful parts of the .NET framework.

## Project 1: Hello, World



Let’s start with a message box that says "Hello, World." It’s very easy.

At the Windows PowerShell ISE command prompt, type (or cut/paste):

New-Label “Hello, World” –Show

This command uses the **New-Label** function in the WPK module. To enlarge the text, use the **FontSize** parameter. The following command increases the font size to 48 point text.

New-Label “Hello, World” –Show -FontSize 48

Very complex items can take a while to complete. Instead of waiting, you can create the items as a background job, and continue using Windows PowerShell for other things while it processes.

To create the "Hello, World" dialog box as a background job, add the **AsJob** parameter.

New-Label “Hello, World” -Show -FontSize 48 -AsJob

The **AsJob** parameter lets you use all of the features of [Windows PowerShell background jobs](http://go.microsoft.com/fwlink/?LinkID=113251), including the **Wait-Job**, **Stop-Job**, and **Remove-Job** cmdlets.

*Hint! Hint!*

* Use tab-completion to avoid those extra keystrokes. Type part of the command and press the <Tab> key. Windows PowerShell tries to complete the command. You can press <Tab> repeatedly to cycle through matching commands.
* To find a cmdlet to create a type of WPF control, such as a button, use the following command format:

get-command -name \*button\* -module wpk

-or-

gcm \*button\* -m wpk

# Project 2: Digital Clock



Now let’s create a simple digital clock. This project includes a UI task that everyone needs to do, which is to run something at a regular interval. We’re going to use events to set that interval, along with commands, to make the clock work.

Here’s the command to create the clock. You can cut/paste it into Windows PowerShell ISE or type it.

New-Label -FontSize 24 -On\_Loaded {

Register-PowerShellCommand -scriptBlock {

$window.Content.Content = (Get-Date | Out-String).Trim()

} -run -in "0:0:0.5"

} -AsJob

This command uses the **New-Label** function in the WPK module. The clock is part of a label, just like the label that contained the "Hello, World" message. The command uses the **FontSize** parameter to set the font size to 24.

New-Label -FontSize 24 -On\_Loaded {

Register-PowerShellCommand -scriptBlock {

$window.Content.Content = (Get-Date | Out-String).Trim()

} -run -in "0:0:0.5"

} -AsJob

The **On\_Loaded** parameter represents the WPF "Loaded" event. When the event occurs, that is, when label is loaded, the command in the **On\_Loaded** script block runs.

New-Label -FontSize 24 -On\_Loaded {

Register-PowerShellCommand -scriptBlock {

$window.Content.Content = (Get-Date | Out-String).Trim()

} -run -in "0:0:0.5"

} -AsJob

In this example, the **On\_Loaded** script block contains the **Register-PowerShellCommand** function. **Register-PowerShellCommand**, a function that is included in the WPK module, runs a specified command in a specified context, such as the window in $window, a context that you'll use often in WPK.

New-Label -FontSize 24 -On\_Loaded {

Register-PowerShellCommand -scriptBlock {

$window.Content.Content = (Get-Date | Out-String).Trim()

}-run -in "0:0:0.5"

} -AsJob

To specify the command and the context, use the **ScriptBlock** parameter of **Register-PowerShellCommand**. In this case, the script block adds the current date and time to the $window context (actually the context of the context).

To get the date and time, use the [Get-Date](http://go.microsoft.com/fwlink/?LinkID=113313) cmdlet. The [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime.aspx) object that **Get-Date** returns is sent to the [Out-String](http://go.microsoft.com/fwlink/?LinkID=113368) cmdlet, which makes the output look like it came from the Windows Powershell console. We use the **Trim** method to remove extra white space.

New-Label -FontSize 24 -On\_Loaded {

Register-PowerShellCommand -scriptBlock {

$window.Content.Content = (Get-Date | Out-String).Trim()

} -run -in "0:0:0.5"

} -AsJob

The **Run** parameter of **Register-PowerShellCommand** runs the command in the script block immediately. The **In** parameter specifies how often the command in the script block runs. In this example, the **Get-Date** command is run every ½ second.

New-Label -FontSize 24 -On\_Loaded {

Register-PowerShellCommand -scriptBlock {

$window.Content.Content = (Get-Date | Out-String).Trim()

} -run -in "0:0:0.5"

} -AsJob

The command creates the new label in a background job, in case it takes extra time to complete.

Now you’ve built a digital clock in a single five-line command.

*Hint! Hint!*

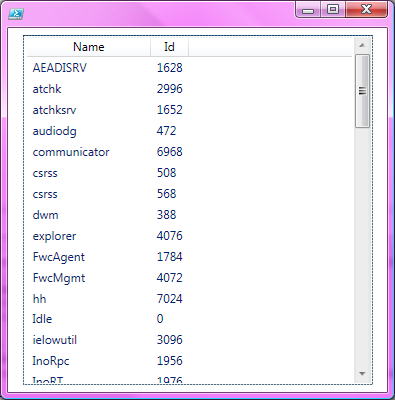
* Every WPF event has a corresponding WPK parameter. The parameter has the same name as the event, but it is preceded by **"On\_"**. For example, the **On\_Loaded** WPK parameter calls the **Loaded** WPF event. To see all of the On\_ parameters of the New-Label cmdlet, type:

get-help new-label -parameter On\_\* | select name

* When you work with the WPK module, you'll be using the following three *automatic* variables quite often. Because they are automatic, you do not have to create them; they are created for you.

|  |  |
| --- | --- |
| $\_ | In Windows PowerShell, $\_ represents the current item. In events, $\_ contains the arguments that are sent to the events. |
| $this | In events, $this is where the event is coming from |
| $window | The window that displays the WPF controls. You can find everything in the window. |

# Project 3: A Process Monitor



Getting the time is one thing, but how about a real-word administrator operation? Let’s create a dashboard for watching processes. This time, we’ll use a few new tricks to check for new processes every 15 seconds and show both the name and the process ID.

Here’s the command that creates the process monitor. You can cut/paste it into Windows PowerShell ISE or type it.

New-ListView -Width 350 -Height 350 -DataBinding @{

ItemsSource = New-Binding -IsAsync -UpdateSourceTrigger PropertyChanged -Path Output

} -View {

New-GridView -AllowsColumnReorder -Columns {

New-GridViewColumn "Name"

New-GridViewColumn "Id"

}

} -DataContext {

Get-PowerShellDataSource -Script {

Get-Process | ForEach-Object { $\_ ; Start-Sleep -Milliseconds 25 }

}

} -On\_Loaded {

Register-PowerShellCommand -Run -In "0:0:15" -ScriptBlock {

$window.Content.DataContext.Script = $window.Content.DataContext.Script

}

} -asjob

This command uses the **New-ListView** function in the WPK module to create a WPF ListView control for the process monitor. The command includes the following parameters.

* The **Width** and **Height** parameters specify the monitor's dimensions.
* The **View** parameter defines the names of the table columns, Name and Id.
* The **DataContext** parameter determines the source of the data that is displayed the table, which is a **Get-Process** command.
* The **On\_Loaded** parameter specifies when the data is retrieved (immediately) and how often it is refreshed (every 16 seconds).
* The **AsJob** parameter runs the command as a background job.

Like any control that can hold more than one item, the ListView has two properties, Items, which contains the actual items, and ItemsSource, which contains a list that can be used to supply items.

To make the user interface run quickly, the command uses the **DataBinding** parameter of **New-ListView**. The **DataBinding** parameter takes a [hash table](http://go.microsoft.com/fwlink/?LinkID=135175) as input. The hash table in this command has only one key/value pair. The key is "ItemsSource" and the value is generated by the **New-Binding** function in the WPK module.

New-ListView -Width 350 -Height 350 -DataBinding @{

ItemsSource = New-Binding -IsAsync -UpdateSourceTrigger PropertyChanged -Path Output

}

The **New-Binding** command uses the **IsAsync** parameter , which means that it populate the items as they come in. The **UpdateSourceTrigger** parameter has a value of **PropertyChanged**, which updates the binding source immediately whenever the binding target property changes. The **Path** parameter indicates that the binding source is the output. Because the **DataBinding** parameter takes a hash table, you can create as many binding as you need.

The **View** parameter of the **New-ListView** function defines the view, which is always a grid view, commonly known as a table. Use the following command format to define the grid view. To specify the names of the columns in the table, use the **New-GridViewColumn** function. The column names, which are case-sensitive, become properties of the GridView object that the **New-GridView** function returns. In this example, there are two columns, **Name** and **Id**.

-View {

New-GridView -AllowsColumnReorder -Columns {

New-GridViewColumn "Name"

New-GridViewColumn "Id"

}

The **DataContext** parameter specifies the data context of the ListView, which is the first place that WPF looks for the path that is specified in a data binding.

The data context is defined by the **Get-PowerShellDataSource** function, which gets data from the WPF controls and sends it to Windows PowerShell asynchronously -- while the control is still running -- so you can see the output of a long-running script while it’s in progress. Get-PowerShellDataSource can be used to execute long-running commands in the background of any user interface. This lets you create a UI that monitors dozens or thousands of machines.

In this example, the data source is a small script that runs the [Get-Process](http://go.microsoft.com/fwlink/?LinkID=113324) cmdlet, but displays its results slowly, sleeping for 25 milliseconds between the display of each process.

-DataContext {

Get-PowerShellDataSource -Script {

Get-Process | ForEach-Object { $\_ ; Start-Sleep -Milliseconds 25 }

}

The **On\_Loaded** parameter of the New-ListView function lets you specify a response to the event that is raised when the ListView is loaded. In this case, it registers a command that sets the Script property of the data context on the window’s content to itself. The **Run** parameter runs the script in the **DataContext** parameter (the **Get-Process** script) as soon as the ListView is loaded and the **In** parameter re-runs the script every 15 seconds to refresh the process list.

-On\_Loaded {

Register-PowerShellCommand -Run -In "0:0:15" -ScriptBlock {

$window.Content.DataContext.Script = $window.Content.DataContext.Script

}

The AsJob parameter runs the command in a background job, so you can work while Windows PowerShell creates the list view for the Process Monitor.

*Hint! Hint!*

* Unlike most items in Windows PowerShell, the names of the columns, which are defined by the **New-GridViewColumn** function, are case-sensitive by default. However, because they are properties of the object, you can find the correct spelling and capitalization by piping the GridView object to the [Get-Member](http://go.microsoft.com/fwlink/?LinkID=113322) cmdlet.

# Project 4: Quick and Easy Media Player

****

Here’s a simple video player built by using the commands in the WPK module. It supports drag and drop, and starts with a random video from public videos:

New-Window -AllowDrop -On\_Drop {

$file = @($\_.Data.GetFileDropList())[0]

$this.Content.Source = $file

$this.Content.Play()

} -On\_Loaded {

$this.Content.Source = dir "$env:PUBLIC\videos\Sample Videos" -Filter \*.wmv |

Get-Random | Select-Object -ExpandProperty Fullname

$this.Content.Play()

} -On\_Closing {

$this.Content.Stop()

} {

New-MediaElement -LoadedBehavior Manual

} -asJob

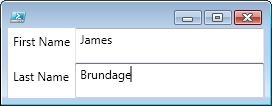
The New-MediaElement function in the WPK module can be used any time you want to play a video, show an image, or play a sound. It’s just a quick example of all of the different things you can do with WPF.

The command uses the New-Window function in the WPK module to create a new window. It uses the following parameters to define the window.

* The **AllowDrop** parameter lets the window accept data that you drag onto it.
* The **On\_Drop** parameter determines what happens when data is dropped onto the window (the Drop event). In this example, the first item is retrieved and played.
* The **On\_Loaded** parameter determines what happens when the window is loaded (the Load event). In this example, a random video (a .wmv file) is selected from the Sample Videos directory and it is played.
* The **On\_Closing** parameter determines what happens when the window is closed (the Closing event). In this example, the media player stops.
* The **AsJob** parameter runs the command as a background job.

The content of the window is provided by the **New-MediaElement** function in the WPK, which creates a WPF control that contains audio or video. The **LoadedBehavior** parameter, which determines the media state whenthe control is loaded, is set to **Manual**, which lets you start and stop the media player.

# Project 5: Dialog Box



C:\PS> $name

Brundage, James

This final project creates a simple dialog box that collects the user's first name and last name and assigns the names to the $name variable. When you close the dialog box, the values that you type in the box are available in the $name variable.

$Name = New-Grid -Rows 2 -Columns 'Auto','1\*' {

$TextChanged = {

$firstName = Get-Resource FirstName | Select-Object -ExpandProperty Text

$lastName = Get-Resource LastName | Select-Object -ExpandProperty Text

$this.Parent.Tag = "$LastName, $FirstName"

}

New-Label "First Name"

New-TextBox -Name FirstName -Column 1 -On\_Loaded {

Set-Resource -Name FirstName -Value $this -Depth -1

} -On\_TextChanged $TextChanged

New-Label "Last Name" -Row 1

New-TextBox -Name LastName -Column 1 -Row 1 -On\_Loaded {

Set-Resource -Name LastName -Value $this -Depth -1

} -On\_TextChanged $TextChanged

} -show

The command uses the **New-Grid** function in the WPK module.

The **Rows** and **Columns** parameter create a 2 x 2 grid. The rows are the same size. The first column is automatically sized and the next column is 1\*, which represent a 1:1 ratio of the available space. As you drag the grid, the textboxes grow to consume most of the available space and the rows spread apart.

The **New-Label** function sets the label for each row. The **New-TextBox** function sets the properties of each text box.

The grid contains a label and a textbox each for the first and last name. When each textbox is loaded (the On\_Loaded parameter of **New-TextBox**), it is set to resource called "FirstName" or "LastName" with a Depth of -1, which means that the objects parent will have a pointer directly to FirstName in its resources property. This means that the other controls will be able to get to FirstName and LastName quickly.

The command uses the **Get-Resource** function to get the First Name and Last Name and store them in the $TextChanged variable. The $TextChanged variable is the value of the **On\_TextChanged** parameter, which responds to the TextChanged event of the new text box.

The text changes update a property called **Tag**. Tag is the first property that WPK will look for when the window is closed. The value of Tag appears in the $name variable when you close the dialog box.

*For more information...*

For more information about WPF, see [Windows Presentation Foundation](http://msdn.microsoft.com/en-us/library/ms754130.aspx) and [System.Windows.Controls](http://msdn.microsoft.com/en-us/library/system.windows.controls.aspx) in MSDN.

For more information about Label controls, see [Labels](http://msdn.microsoft.com/en-us/library/ms743463.aspx) and [Label Class](http://msdn.microsoft.com/en-us/library/system.windows.controls.label.aspx) in MSDN.

For more information about ListView controls, see [ListView](http://msdn.microsoft.com/en-us/library/ms754239.aspx) and [ListView Class](http://msdn.microsoft.com/en-us/library/system.windows.controls.listview.aspx) in MSDN.

For more information about GridView controls, see [GridView Overview](http://msdn.microsoft.com/en-us/library/ms752213.aspx) and [GridView Class](http://msdn.microsoft.com/en-us/library/system.windows.controls.gridview.aspx) in MSDN.

For more information about data binding, see [Binding Class](http://msdn.microsoft.com/en-us/library/system.windows.data.binding.aspx), [Data Binding Overview](http://msdn.microsoft.com/en-us/library/ms752347.aspx), and [Optimizing Performance: Data Binding](http://msdn.microsoft.com/en-us/library/bb613546.aspx) in MSDN.

For more information about Window controls, see [WPF Windows Overview](http://msdn.microsoft.com/en-us/library/ms748948.aspx) and [Window Class](http://msdn.microsoft.com/en-us/library/system.windows.window.aspx)

This Quick Start topic should have given you a quick idea of all that’s possible with WPK. To explore more, check out the examples subdirectory of the WPK module, or just type "Get-Help" followed by any command name.

Hope this Helps,

James Brundage [MSFT]