



# PowerShell Studio - Help Manual

---

© 2020 by SAPIEN Technologies Inc., all rights reserved

<b>1. Welcome to PowerShell Studio</b>	<b>6</b>
<b>2. Introduction</b>	<b>7</b>
About PowerShell Studio .....	7
How to Buy PowerShell Studio .....	7
<b>3. Getting Started</b>	<b>9</b>
Installing PowerShell Studio .....	9
Staying Up-to-date .....	15
Getting Help .....	15
<b>4. Basic Orientation</b>	<b>18</b>
User Interface .....	18
The Start Page .....	19
The Ribbon .....	20
Quick Access Toolbar .....	24
Panels and Docking .....	24
Status Bar .....	27
Customizing Your Workspace .....	29
Selecting a Style .....	29
Customizing the Quick Access Toolbar .....	29
Panels and Layouts .....	30
Introduction to Panels .....	30
Working with Panels .....	32
Layouts .....	42
<b>5. Script Editor</b>	<b>46</b>
Editing Aids .....	46
Line Numbering and Visual Features .....	46
Code Folding .....	47
Reference Highlighting .....	50
Syntax Checking .....	51
Navigation and Bookmarks .....	52
Navigation Bar .....	54
Clipboard Integration .....	59
Find and Replace Options .....	60
Find and Replace .....	60
Find in Files .....	62
Find All References .....	66

<b>PrimalSense™ .....</b>	<b>68</b>
<b>Converting Cmdlets and Aliases .....</b>	<b>75</b>
<b>Snippets .....</b>	<b>78</b>
<b>Script Signing .....</b>	<b>81</b>
<b>File Encoding .....</b>	<b>83</b>
<b>Context Menu Options .....</b>	<b>83</b>
<b>Functions and Parameters .....</b>	<b>85</b>
<b>Function Builder .....</b>	<b>85</b>
Function (Cmdlet) Name .....	88
Synopsis and Description .....	88
Cmdlet Binding .....	88
Output Type .....	89
Parameter Sets .....	90
Default Parameter Set .....	91
Parameters .....	91
Parameter Set Filter .....	95
Parameter Editor .....	97
Special Considerations .....	102
Create Functions from Selection .....	103
Editing Functions .....	104
Importing Functions .....	106
Parameter Builder .....	108
<b>Comment-Based Help .....</b>	<b>110</b>
Comment-Based Help Templates .....	114
About the Comment-Based Help Template .....	114
Comment-Based Help Template Variables .....	115
Creating a Comment-Based Help Template .....	116
Selecting an Existing Comment-Based Help Template .....	117
Multi-line or Single-line Comments .....	118
<b>File Type Templates .....</b>	<b>119</b>
Using Predefined File Templates .....	119
Creating New File Templates .....	122
Template Variables .....	126
<b>Rename Refactoring .....</b>	<b>128</b>
<b>Verifying Your Script .....</b>	<b>133</b>
<b>6. Running and Debugging Scripts .....</b>	<b>135</b>
<b>Run and Debug Ribbon Controls .....</b>	<b>135</b>
<b>Running Scripts .....</b>	<b>141</b>
<b>Debugging Scripts .....</b>	<b>142</b>
Working with Breakpoints .....	142

Working with Tracepoints .....	144
Passing Parameters .....	147
Debug Panels .....	148
Running and Debugging Remotely .....	148
<b>7. GUI Designer</b>	<b>152</b>
Forms Designer Introduction .....	152
Creating a New Form .....	156
Working with Form Controls .....	159
Preview GUI .....	165
Adding Events .....	166
Form Templates .....	171
Using Predefined Form Templates .....	171
Creating New Form or Grid Templates .....	174
Working with Grid Templates .....	178
Exporting Form Scripts .....	181
Initializing GUI Controls .....	182
Control Helper Functions .....	184
Property Sets .....	188
Control Sets .....	193
<b>8. Panels</b>	<b>204</b>
Call Stack Panel .....	206
Console Panel .....	206
Debug Console .....	208
Find Results Panel .....	209
Function Explorer Panel .....	210
Help Panel .....	212
Object Browser .....	214
Output Panel .....	221
Performance Panel .....	223
Project Panel .....	225
Project Files and Folders .....	229
Properties Panel .....	240
Snippets Panel .....	245
Toolbox Panel .....	250
Tools Output Panel .....	254

Variables Panel .....	256
Watch Panel .....	259
<b>9. Projects</b>	<b>262</b>
Project Templates .....	262
Available Project Templates .....	262
Creating a Project .....	265
Collection Project .....	268
Module Project .....	271
New Module from Functions .....	272
Project Properties .....	276
Managing Project Files .....	278
Project File Properties .....	280
Adding Script Parameters to Projects .....	281
Running a Project .....	282
Exporting a Project .....	283
Form Return Variables .....	284
Projects and Source Control .....	287
<b>10. Packaging Scripts</b>	<b>289</b>
Creating a Script Package .....	289
Setting up the Script Packager .....	290
<b>11. Source Control Integration</b>	<b>302</b>
Universal Version Control .....	302
Microsoft Source Code Control Integration .....	305
<b>12. ScriptMerge</b>	<b>312</b>
Running ScriptMerge .....	312
Comparing Files .....	312
Comparing Folders .....	315
Comparing Groups .....	317
Context Menu Options .....	318
Navigating Between Differences .....	319
Reconciling Differences .....	320
Signing Scripts .....	320
ScriptMerge Settings .....	321

<b>13. Snippet Editor</b>	<b>326</b>
<b>14. Options and Settings</b>	<b>332</b>
Accessing the Options .....	332
General .....	332
Backup .....	336
Console .....	338
Debugger .....	341
Designer .....	343
Editor .....	347
Assemblies .....	356
Formatting .....	357
PrimalSense™ .....	366
Panels .....	368
PowerShell .....	371
Source Control .....	372
<b>15. Remote Script Execution Engine</b>	<b>376</b>
<b>16. Reference</b>	<b>382</b>
SAPIEN Updates .....	382
Keyboard Shortcuts .....	385
Appendices .....	395
Appendix A: Manual Version .....	396
Appendix B: Icon License Attribution .....	396

## 1 Welcome to PowerShell Studio

### Windows PowerShell scripting and tool-making has never been easier!

---



Welcome to PowerShell Studio, the premier scripting and tool-making environment for Windows PowerShell. This single tool will meet all your scripting needs. In addition to creating graphical tools using Windows PowerShell with the GUI designer, you can also create Windows PowerShell script modules in minutes and easily convert your existing functions to a distributable module.

PowerShell Studio features a robust editor and a script packager with advanced option and platform selections to help you deliver solutions targeted at specific environments.

### About this documentation

---

This help is designed to show you how to use PowerShell Studio—you can do a quick overview to get started, work through the topics in detail, and refer back to this guide for additional information when needed.

### Getting started - new users

---

- [Download and install PowerShell Studio](#).
- Get a quick [overview of the user interface](#)<sup>[18]</sup> and see how to [customize your workspace](#)<sup>[29]</sup>.
- Learn how to use PowerShell Studio's powerful [script editor](#)<sup>[46]</sup>, [run and debug scripts](#)<sup>[135]</sup>, and create [GUI forms](#)<sup>[152]</sup>.
- Visit the [support forum](#) to get help from SAPIEN staff and other experienced PowerShell Studio users.

## 2 Introduction

This section provides an overview of the PowerShell Studio features, shows you how to purchase directly online or through a reseller, and lets you know how to get answers to your questions.

### 2.1 About PowerShell Studio

PowerShell Studio is the premier Windows PowerShell integrated scripting and tool-making environment.

#### Key Features

---

- Fully-featured **PowerShell Editor**.
- Visually create **PowerShell GUI** tools.
- Convert scripts into **executables (.exe)** files.
- Create **MSI installers**.
- Create **modules, advanced functions, and windows services**.
- For a complete list of current features, visit the [PowerShell Studio product page](#).

#### What's New

---

We are always updating and improving PowerShell Studio. You can learn about the latest product updates on our blog and in the release build log.

- Check out the latest PowerShell Studio tips and product feature demonstrations on the [SAPIEN blog](#).
- View a brief synopsis of what was changed, added, or fixed in the most recent PowerShell Studio build in the product [version history](#).
- Submit [feedback and suggestions](#).

### 2.2 How to Buy PowerShell Studio

You can buy PowerShell Studio online with all major credit cards. As soon as your transaction completes, you will be able to [download and install](#) the program.

For answers to your pre-order questions, check out the [SAPIEN Frequently Asked Questions](#) or post in the [Trial Software / Pre-sales Technical Questions](#) forum.

## Order link and PowerShell Studio product page

---

**Online orders:**

<https://www.sapien.com/store/powershell-studio>

**Worldwide authorized resellers:**

<https://www.sapien.com/company/resellers>

**PowerShell Studio product page:**

[https://www.sapien.com/software/powershell\\_studio](https://www.sapien.com/software/powershell_studio)

## 3 Getting Started

This section shows you how to download and install PowerShell Studio, how to keep the application updated with the latest builds, and how to find additional help.

### 3.1 Installing PowerShell Studio

This section shows you how to install and activate PowerShell Studio, and it also covers how to remove your activation if you need to use it on a different computer.

#### Downloading PowerShell Studio

---

All SAPIEN Technologies software products are downloadable only. Download registered products from your [SAPIEN Account Registered Products page](#).

Select the 64-bit version of PowerShell Studio to download. The installer software will save to your default download folder (e.g., *SPS20Setup\_5.7.171\_010720\_x64*).

**i** Starting with the PowerShell Studio 2020 product release, 32-bit versions are no longer available. Current owners of a license that includes a 32-bit product will have access to that from their [SAPIEN Account Registered Products page](#).

Want to try before you buy? You can [download a trial version here](#).

#### Installing PowerShell Studio

---

Follow these instructions to install PowerShell Studio.

##### How to install PowerShell Studio

---

1. In your default download folder, double-click on the downloaded program (e.g., *SPS20Setup\_5.7.171\_010720\_x64*).
2. Reply **Yes** to the "Do you want to allow this app to make changes to your device?" prompt.

The installation wizard will first check several items, such as available disk space and the presence of previous builds. If the environment is adequate, the installer will display the legal agreement which you must accept to proceed:

- a. **Read** the terms of the license agreement.
- b. **Accept** the terms of the license agreement. You should never accept license terms unless you have read them, and you understand them.
- c. Once you have accepted the terms, click **Install**.

**i** The software will install in the default location as shown, unless you change the path.



3. The installation may take several minutes.



- When PowerShell Studio successfully completes the installation, click **Finish**.



## Silent Installation

Use this command if you need to install silently: SPSxxSetup\_x.x.xxx\_xxxxxx\_x64.exe /exenoui /qn  
(e.g., SPS20Setup\_5.7.171\_010720\_x64 /exenoui /qn)

## Troubleshooting Installation

If you encounter any problems installing PowerShell Studio please report them in the [Installation Issues support forum](#).

Use these Installer Log parameters to output to a log file: Installer.exe /exenoui /qn /L\*v .\SPS\_Install.log

## Firewall Considerations

---

PowerShell Studio can be configured to install a small service to support the Remote Script Execution Engine (RSEE). This will result in a firewall warning as the service attempts to open the port it listens on. *For information on configuring RSEE see [Remote Script Execution Engine](#)* [376].

PowerShell Studio occasionally attempts to access a text file located on the [sapien.com](#) web site. This text file contains the version number of the current version of PowerShell Studio—your firewall software may warn you when PowerShell Studio attempts to read this file for the first time. PowerShell Studio does not transmit any personally-identifiable information when making this check—its sole purpose is to notify you when updates are available.

PowerShell Studio also accesses the web to activate the product (after initial installation), and to display web pages when you click on web links within the application.

## Activating and Deactivating PowerShell Studio

---

### Product Activation

---

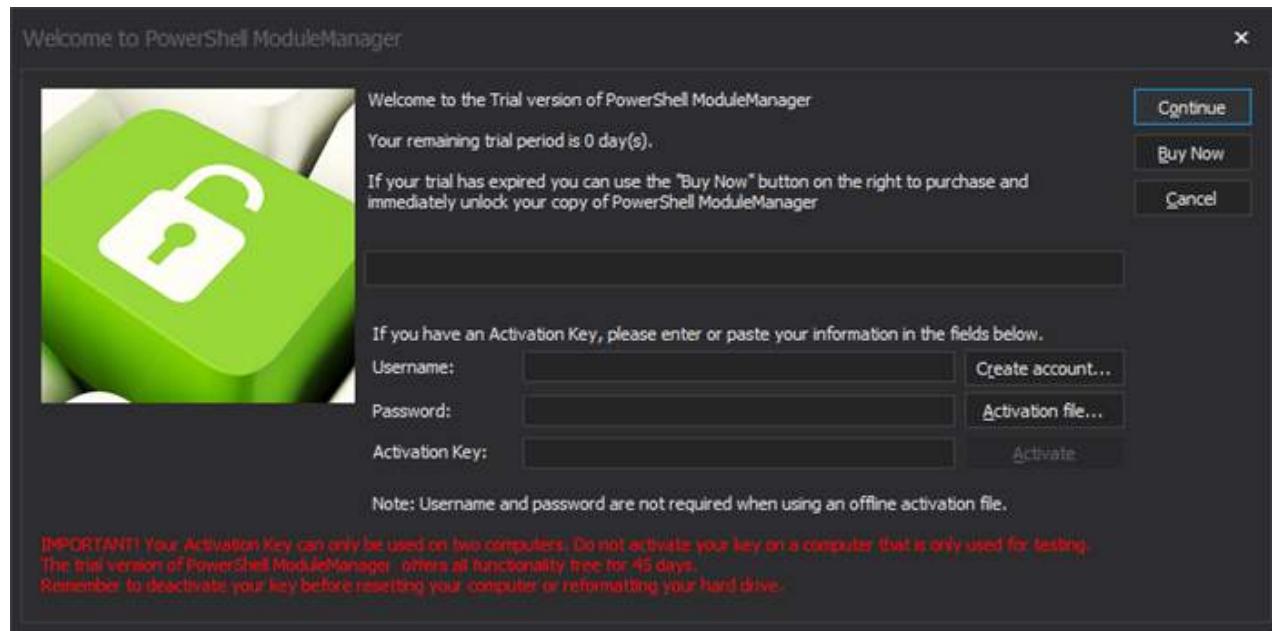
Registration is required to activate and operate the product, and also to obtain any customer service or technical support benefits. Registration only takes a few moments to complete and provides you with access to special offers including preferred pricing on renewals. *You will need an active internet connection to complete product registration.*

An active internet connection may not be required if you have a legitimate reason for needing [offline access](#). To request offline activation [please fill out this request](#). All requests are considered on a case-by-case basis.

#### To activate PowerShell Studio

---

The first time you launch a SAPIEN product, the Welcome screen is displayed.



The steps to activate the product vary depending on whether or not you already have a SAPIEN account.

👉 Follow the steps in the [Quick Guide to SAPIEN Software Activation](#) to activate the software.

If you are unable to activate the product, contact [sales@sapien.com](mailto:sales@sapien.com).

## Product Deactivation

As outlined in our [End-User License Agreement](#), each single-user activation key is entitled to a maximum of two devices to be activated and operating at any given time. You may deactivate your devices to free up your activations at your own leisure, but there are also certain circumstances where proper deactivation is crucial in order to prevent the loss of your allotted two activations.

❗ Uninstalling the software from your device does **not** deactivate the activation key.

### To deactivate your activation key

In the top-right of PowerShell Studio above the ribbon, click the gold certificate button.



The Activation Information window will open.

👉 Follow the steps in the [SAPIEN Software Activation / Deactivation FAQ](#) to deactivate your activation key.

## 3.2 Staying Up-to-date

We are continually updating PowerShell Studio, both to remove bugs and to add and improve product features. We recommend always staying current with the most recent version to ensure that you are taking advantage of the latest features, functionality, and product stability.

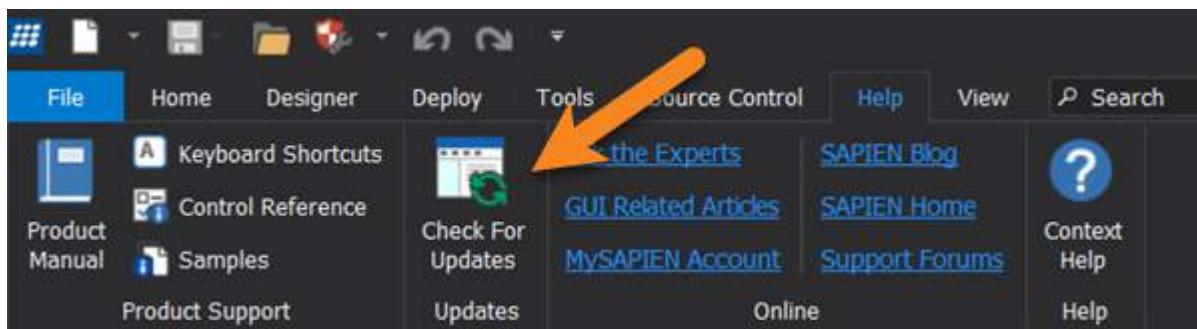
- The details for every PowerShell Studio release are available in the [version history](#).

### Check for Updates

By default, PowerShell Studio will automatically check for software updates. You can also manually check for updates.

#### To check for updates

- On the Help ribbon > in the Updates section, click **Check For Updates** to open the [SAPIEN Updates](#) tool and see if there is a new PowerShell Studio build available:



## 3.3 Getting Help

This help manual has been designed to provide all the information you will need for using PowerShell Studio. In addition to the information in this guide, you can also ask questions in the [online support forums](#).

- View PowerShell Studio product feature demonstrations and release details on [our blog](#).

### Displaying the help manual

- On the Help ribbon > in the Product Support section, click **Product Manual**.

### User forums and support

SAPIEN Technologies provides a variety of ways to get help with PowerShell Studio, including community support forums for your scripting questions.

## Support Options

---

Every registered PowerShell Studio activation key under active maintenance includes basic support in our [PowerShell Studio product support forum](#).

### Premium Support

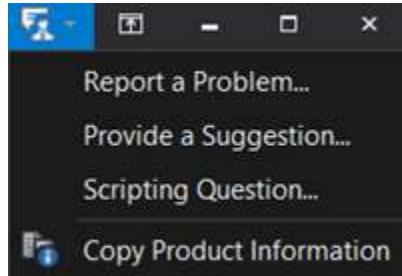
SAPIEN also offers [Premium Support](#), an elevated support option, at an additional cost. Premium Support gives you access to our direct technical ticketing system and guarantees a response within 24 hours, as well as personalized attention until the issue is resolved.

- If your PowerShell Studio maintenance has expired, in order to obtain support you must [renew](#) or [reinstate](#).

## Support Forums

---

The **Send Feedback** menu on the top-right of the ribbon header provides direct links to our support options:



### PowerShell Studio Forums

SAPIEN provides product support forums where our development team answers user questions. Our support technicians monitor the forums daily, but response times are not guaranteed.

The following PowerShell Studio support options are available on the **Send Feedback** menu:

- **Report a Problem...**  
Opens the [PowerShell Studio forum](#) where you can report a problem with the software or ask a product-specific question.
  - **Provide a Suggestion...**  
Opens the [Feature Request](#) page on the SAPIEN site where you can make a product feature request or suggestion.
- You will need to provide your [PowerShell Studio and OS version information](#)<sup>17</sup> to obtain support.

### Scripting Forums

The following scripting and programming support options are available on the **Send Feedback** menu:

- **Scripting Question...**

Opens the [Scripting Answers](#) forums where you can access community support for answers to your scripting questions.

#### ***Product Version Information***

To report a problem in the [PowerShell Studio forum](#), you will need to provide your SAPIEN product and OS version information.

- **Copy Product Information**

Copies the product version information to your clipboard.

## How to copy version information

---

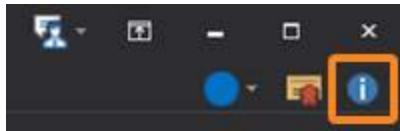
To report a problem in the [PowerShell Studio forum](#), you will need to include the product version and build, and also your OS version and build—and indicate 32 or 64-bit for each.

#### ***To copy the required version information***

---

1. On the Send Feedback menu > select **Copy Product Information**.
2. Paste the version information into your [PowerShell Studio forum](#) post.

👉 You can also copy the version information by clicking the About button in the top-right of the PowerShell Studio workspace, and then clicking Copy Version Info:



# Basic Orientation

## 4 Basic Orientation

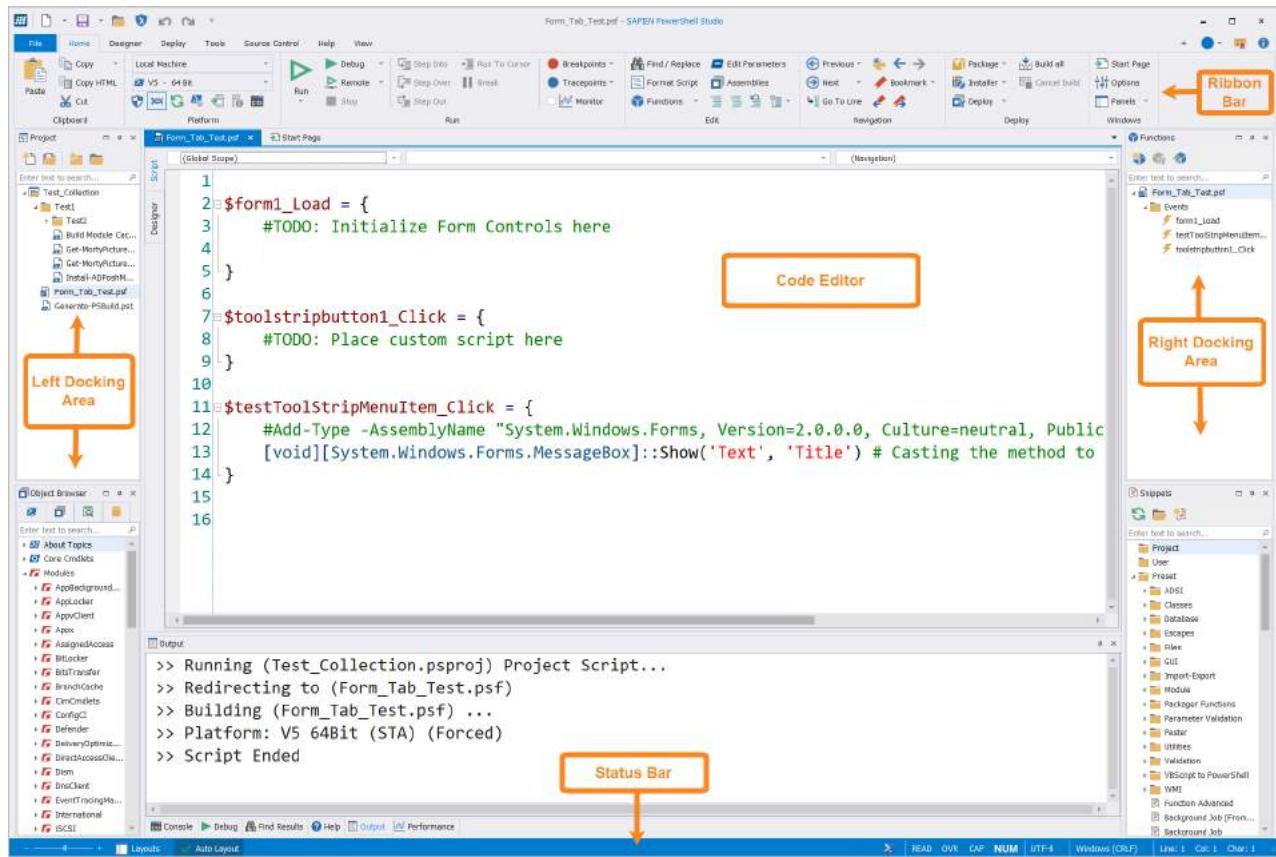
The PowerShell Studio window has many useful features and can be easily customized for various tasks. This section explains basic navigation by introducing you to the user interface and showing you how to customize your workspace.

### 4.1 User Interface

This section provides a basic introduction to some of the main PowerShell Studio user interface elements.

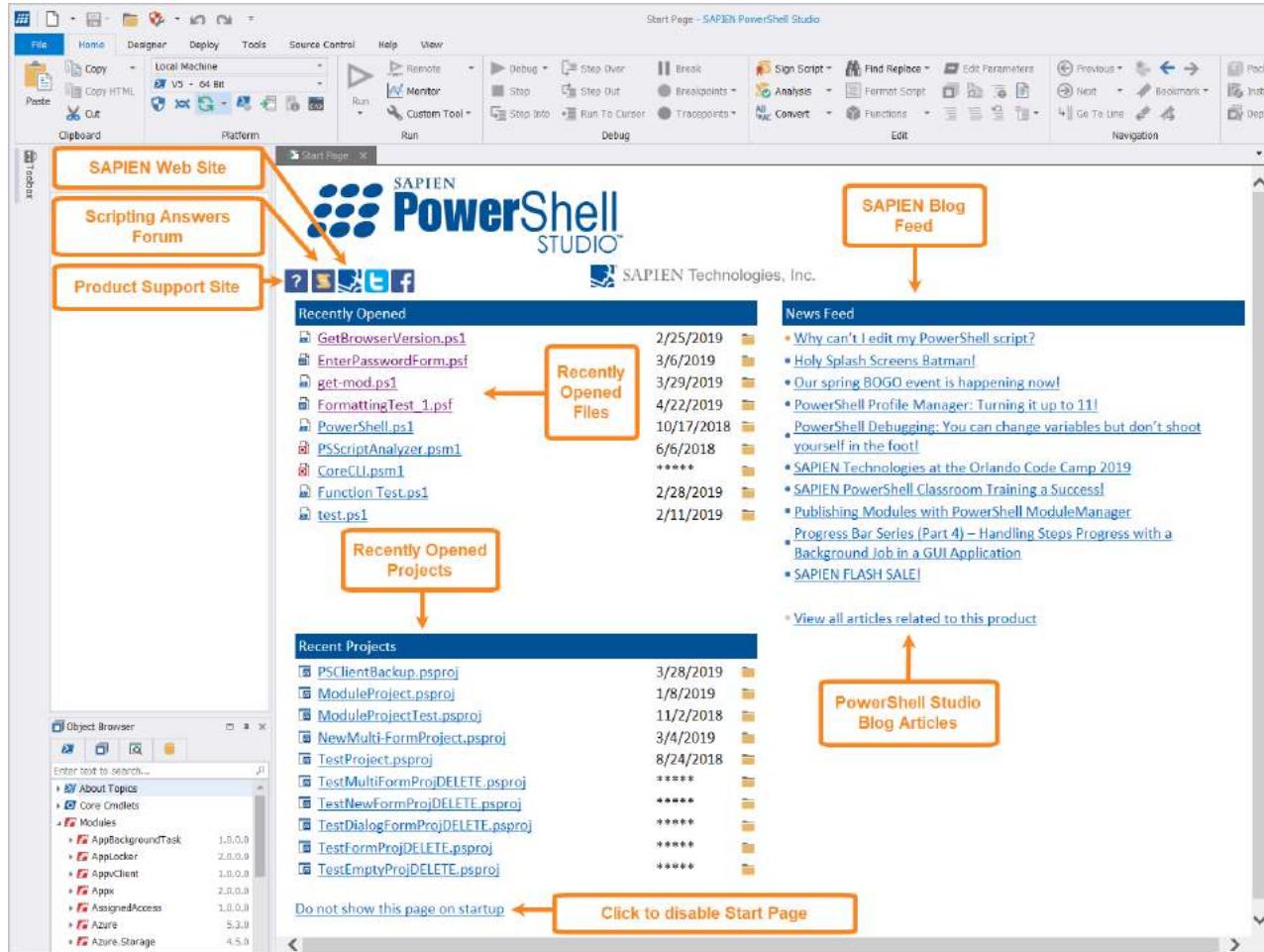
### PowerShell Studio Program Window

The image below shows the major features of the customizable PowerShell Studio window:



## 4.1.1 The Start Page

When you start PowerShell Studio, the Start Page opens in the center of the window:



### Start Page Links

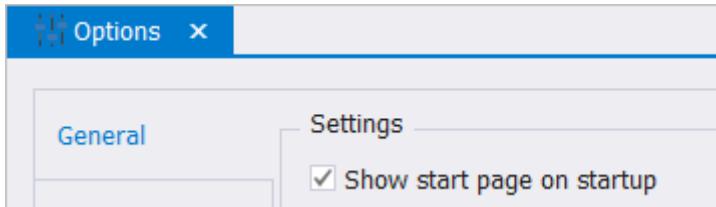
-  [SAPIEN's Product Support Web Site](#)
-  [SAPIEN's Scripting Answers Web Forum](#)
-  [SAPIEN's Web Site](#)
-  [SAPIEN's Twitter Feed](#)
-  [SAPIEN's Facebook Page](#)
- [Recently Opened Files](#)
- [Recently Opened Projects](#)
- [SAPIEN's Blog News Feed](#)

## How to disable the Start Page

- Click the **Do not show this page on startup** link at the bottom-left of the Start Page.

## How to enable the Start Page

- Go to **File > Options > General > Settings** and check **Show start page on startup**:



## 4.1.2 The Ribbon

PowerShell Studio displays a ribbon bar at the top of the application window. This topic covers the different [ribbon tabs](#)<sup>[20]</sup> and also the [ribbon header buttons](#)<sup>[23]</sup> located on the right above the ribbon.

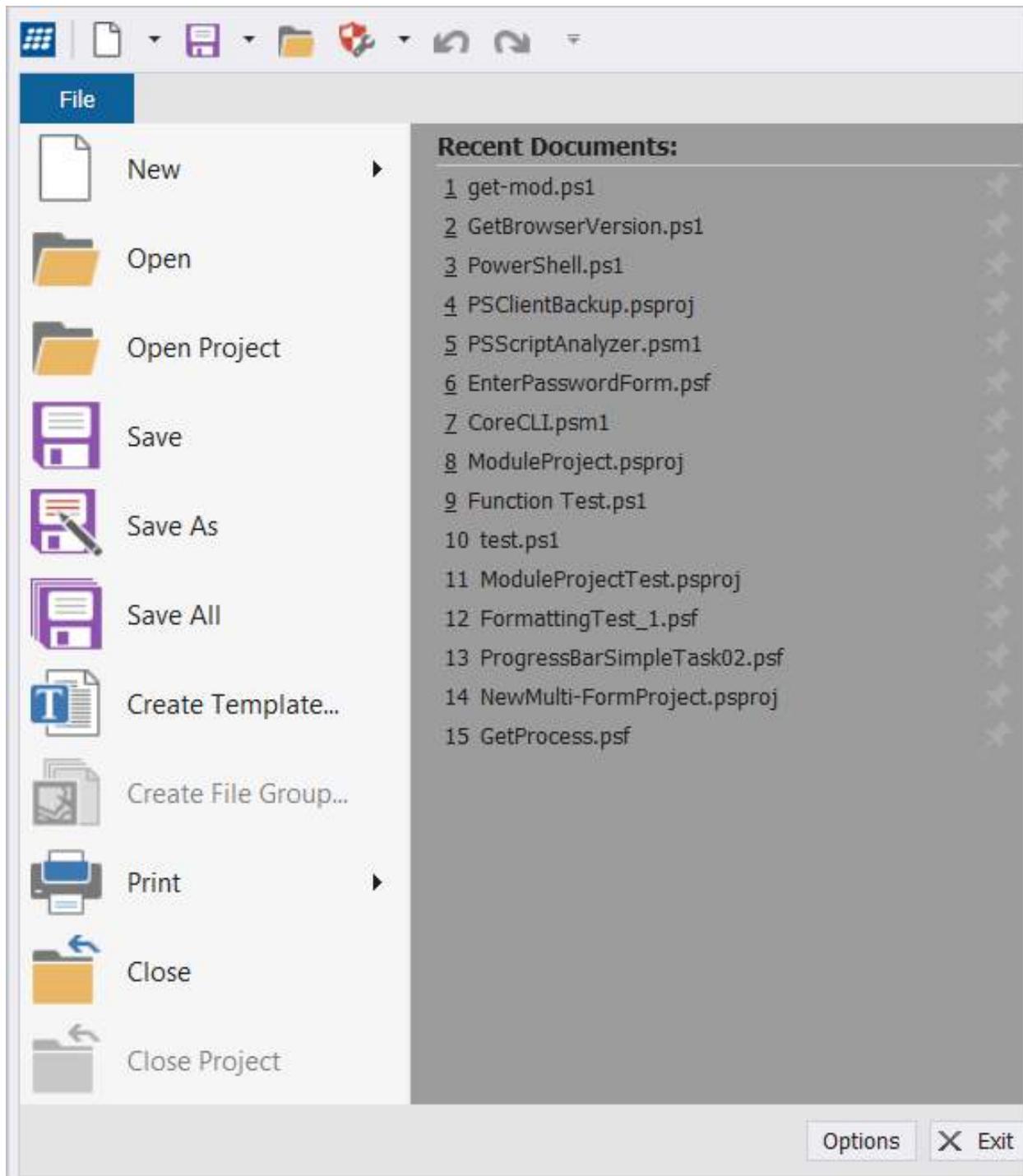
### Ribbon Tabs

The tabs on the ribbon bar are:

- **File**

The File tab contains functions related to opening and closing files and projects, and printing:

# Basic Orientation



- **Home**

The Home tab contains the functions used most often while scripting and is the default tab in PowerShell Studio:



# Basic Orientation

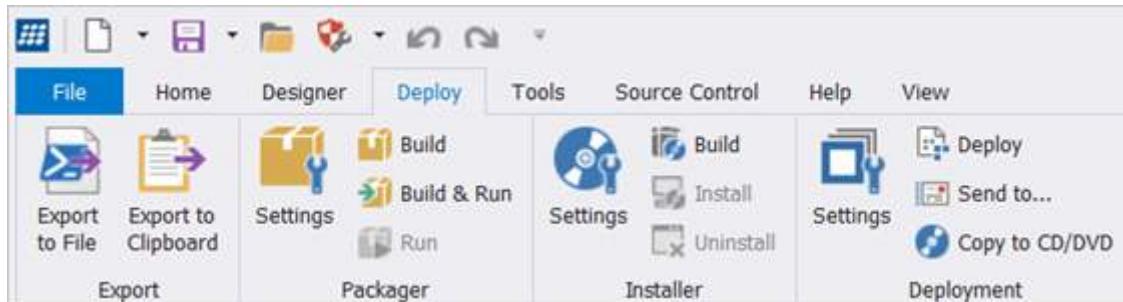
- **Designer**

The Designer tab contains functions related to forms manipulation and creation:



- **Deploy**

The Deploy tab contains tools to create packaged executables, MSI installers and deployment procedures:



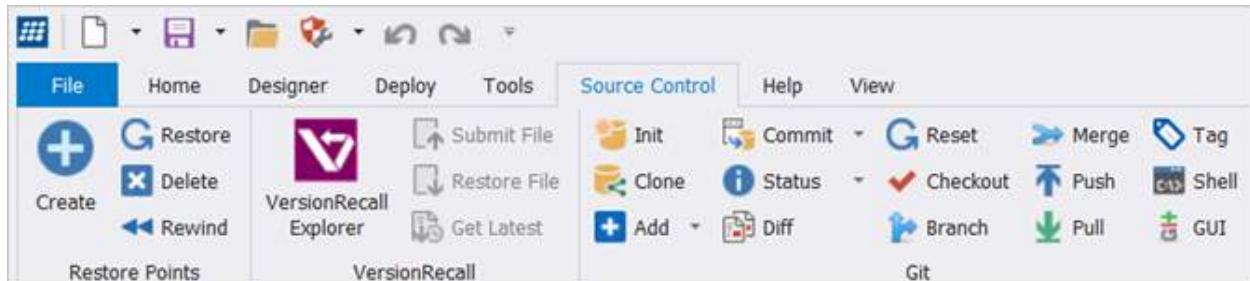
- **Tools**

The Tools tab contains links to external programs, syntax checking, script signing, compare files and more:



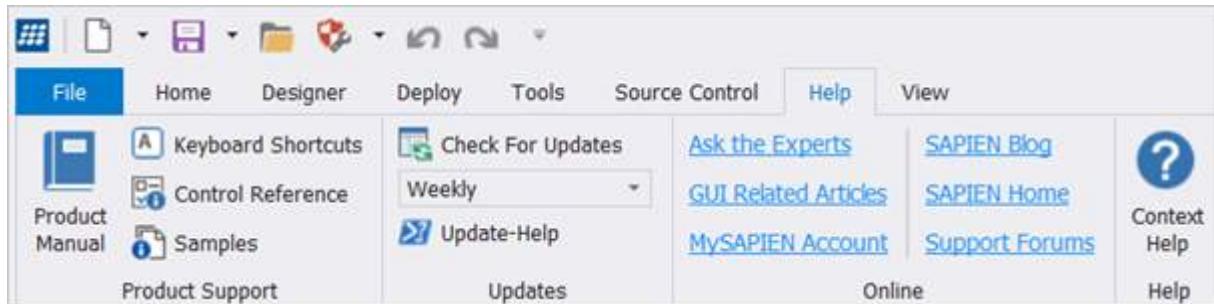
- **Source Control**

On the Source Control tab you can access restore points, VersionRecall, and other third party source control commands such as Git:



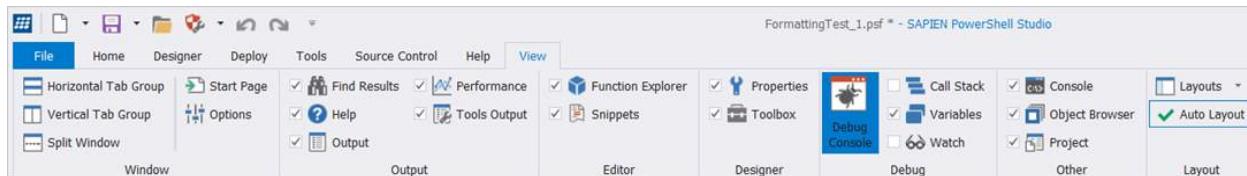
- **Help**

The Help tab contains links to product updates, the manual, and online links to the SAPIEN website and support forums:



- **View**

The View tab lets you quickly toggle panels open and closed, split the Editor window, change layouts, and more:



## Ribbon Header Buttons

There are also four ribbon header buttons on the right above the ribbon bar:

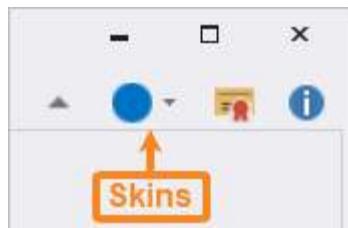


From left to right:

- **Minimize the Ribbon** (*Ctrl+F1*) - Only show tab names on the ribbon.
- **Skins** - [Select color themed skins](#) [23].
- **License Information** - Display and edit your current license information.
- **About** - Display product information.

## Color Themed Skins

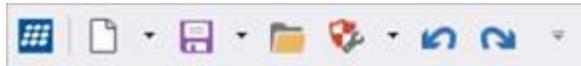
The main color theme used by PowerShell Studio can be changed from the Skins button on the top-right of the window. There are a variety of color themes to choose from:



## 4.1.3 Quick Access Toolbar

The Quick Access Toolbar on the top-left of the program window provides direct access to frequently used functions.

### Quick Access Toolbar - Buttons



From left to right:

- **New (Ctrl+N)** - Create a new document.
- **Save (Ctrl+S)** - Save the current document.
- **Open (Ctrl+O)** - Open an existing document.
- **Execution Policy** - Opens the Script Security Center where you can change PowerShell's execution policy.
- **Undo (Ctrl+Z)** - Undo the previous modification.
- **Redo (Ctrl+Y)** - Redo the last undone modification.
- **Customize** - Customize the Quick Access Toolbar.

Click the drop-down arrow to:

- Toggle each button to Display or Not Display on the toolbar.
  - Show the Quick Access Toolbar below the ribbon.
- Add your own frequently used functions to the Quick Access Toolbar.

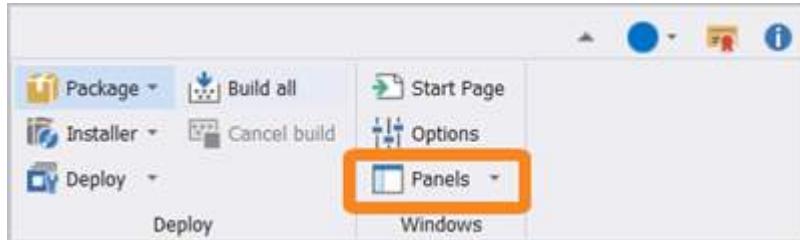
## 4.1.4 Panels and Docking

Some PowerShell Studio features have dedicated dockable window *panels*. You can open, close, dock, and undock the panels, and save your panel arrangements in a *layout*. This topic provides an introduction to the PowerShell Studio panels by showing you how to access the available panels and how to adjust them.

## Available Panels

### How to access the list of available panels

- Select the Home tab > in the Windows section, select Panels:



The panels available in PowerShell Studio:

	<u>Call Stack</u>	Ctrl+Alt+P, K
	<u>Console</u>	Ctrl+Alt+P, C
	<u>Debug Console</u>	Ctrl+Alt+P, D
	<u>Find Results</u>	Ctrl+Alt+P, R
	<u>Function Explorer</u>	Ctrl+Alt+P, F
	<u>Help</u>	Ctrl+Alt+P, H
	<u>Object Browser</u>	Ctrl+Alt+P, B
	<u>Output</u>	Ctrl+Alt+P, O
	<u>Performance</u>	Ctrl+Alt+P, M
	<u>Project</u>	Ctrl+Alt+P, J
	<u>Properties</u>	Ctrl+Alt+P, P
	<u>Snippets</u>	Ctrl+Alt+P, S
	<u>Toolbox</u>	Ctrl+Alt+P, T
	<u>Tools Output</u>	Ctrl+Alt+P, L
	<u>Variables</u>	Ctrl+Alt+P, V
	<u>Watch</u>	Ctrl+Alt+P, W

For details about the content of each panel, see *Panels*.

## Working with Panels

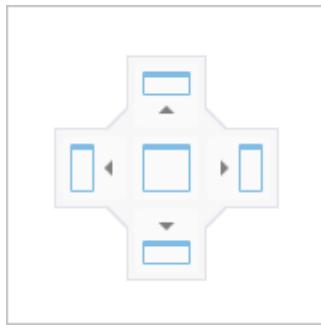
Panels can be adjusted by using the buttons at the top-right of the panel:



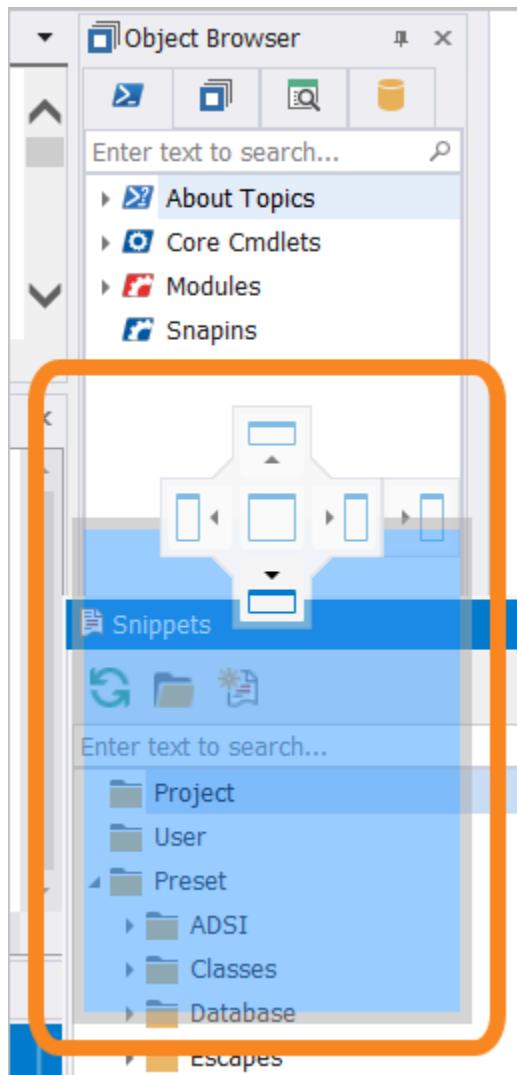
## From left to right

- **Maximize / Restore** – This option is visible when there is more than one panel in a location. You can Maximize the panel to full-size in the location, or Restore the panel so that all panels in the location are visible.
- **Auto Hide / Dock** – Auto Hide will vertically collapse, or "Hide" the panel in the location. Dock will open the panel to a fixed position in the location.
- **Close** – Remove the panel from the location.

You can left-click on the title bar of a panel and drag it to another location. The docking indicator will appear as soon as you begin to drag the panel:



You can drop the panel onto a docking hint to have it snap to the desired location, or leave the panel floating in the window. As shown by the blue highlighting in the example below, the Snippets panel will be docked below the Object Browser:



## 4.1.5 Status Bar

The status bar at the bottom of the PowerShell Studio window displays information about the current configuration of PowerShell Studio.

The left side of the status bar contains the following:



- **Font Size Slider**

The Font Size Slider is used to increase or decrease the font size used in the PowerShell Studio editor window.

- **Layouts**

The Layouts menu is used to define and choose different layouts for the scripting environment.

- **Auto Layout**

The Auto Layout button is used to enable or disable context sensitive layout changes.

To learn more about the layout options, see [Layouts](#)<sup>[42]</sup>.

## Indicators

The right-side of the status bar includes the following status indicators:



- **Running Script**  
Appears when a script is running.
- **Refreshing Local Cache**  
Appears when the local cache is refreshing.
- **PowerShell Detected**  
Appears if PowerShell is detected.
- **READ**  
Read-only status of current file.
- **OVR**  
Overwrite.
- **CAP**  
Caps Lock.
- **NUM**  
Num Lock.
- **File Encoding**  
You can change the encoding of a file by selecting an option from the file encoding menu on the status bar: ASCII, UTF-8-BOM, UTF-8, UTF-16 BE, UTF-16 LE.
- **Line Terminator**  
You can change the line terminator by selecting an option from the line termination menu on the status bar: Windows (CRLF), Unix (LF).
- **Line**  
The line position of the caret in the current file.
- **Col**  
The column position of the caret in the current line.
- **Char**  
The character position of the caret in the current line.

When a script is running or the local cache is refreshing, the relevant icon appears on the status bar.

To cancel a running process or task, click the appropriate icon and select **Stop**.



## 4.2 Customizing Your Workspace

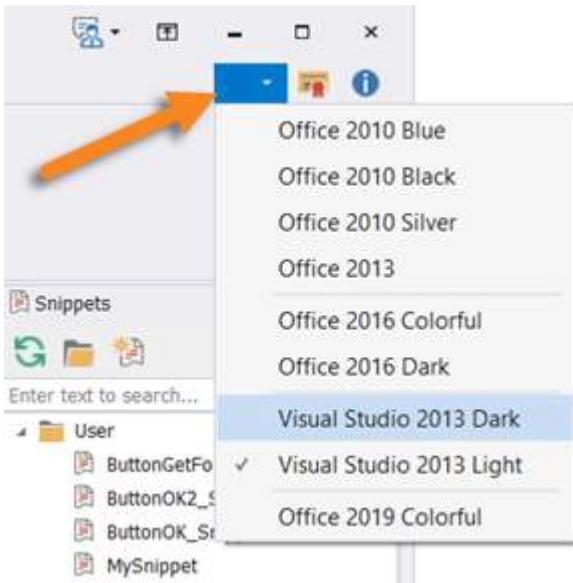
The PowerShell Studio workspace can be easily customized to suit your personal preference.

### 4.2.1 Selecting a Style

A style is a visual layout, skin, or theme.

#### To change the style for PowerShell Studio

- In the upper right corner, click **Skins** and then select a style:



**i** The default style is Visual Studio 2013 Light.

### 4.2.2 Customizing the Quick Access Toolbar

The Quick Access Toolbar at the top-left of the PowerShell Studio program window provides access to your most frequently used tools.

#### To add buttons to the Quick Access Toolbar

- Navigate to the desired function, then right-click and select **Add to Quick Access Toolbar**.

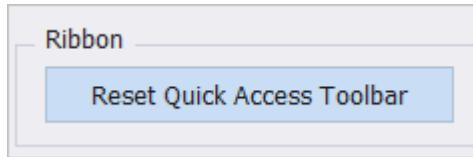
For example, to add a Keyboard Shortcuts button to the Quick Access Toolbar:

- On the Help ribbon, right-click **Keyboard Shortcuts** > select **Add to Quick Access Toolbar**.



## To reset the Quick Access Toolbar

- The Quick Access Toolbar can be reset on the Panels tab of the Options dialog (File > Options > Panels or Home > Options > Panels):



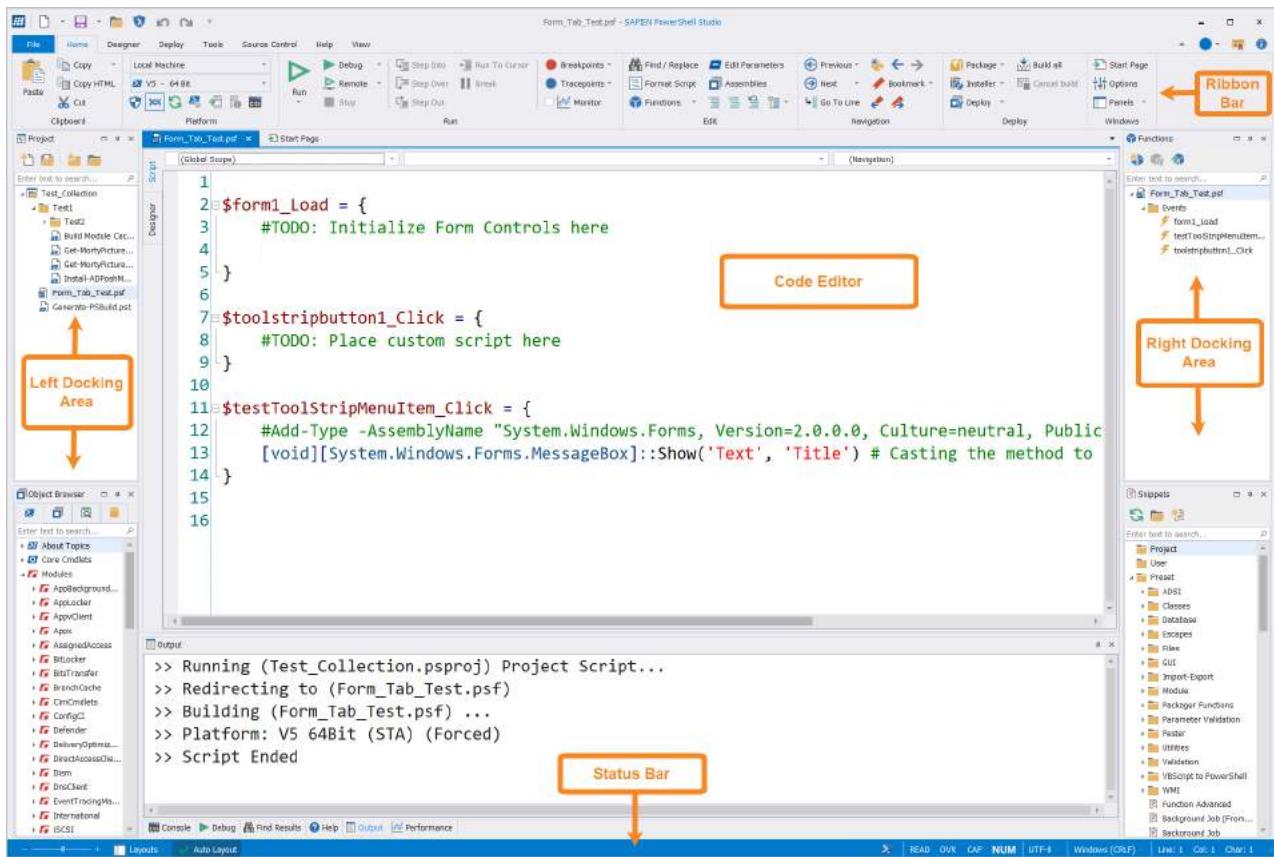
## 4.2.3 Panels and Layouts

Some PowerShell Studio tools are displayed in dockable window panels. This section explains how to work with panels, and how to save your panel arrangements in a layout.

### 4.2.3.1 Introduction to Panels

You will work with panels in the customizable PowerShell Studio window:

# Basic Orientation

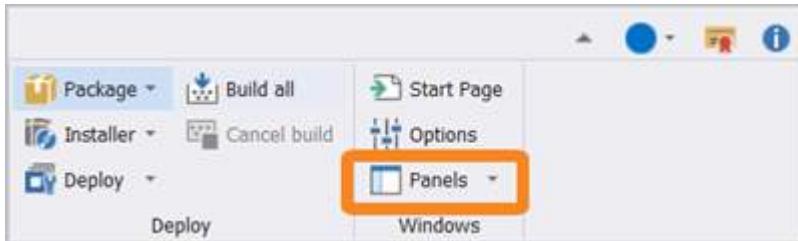


You can open, close, dock, and undock panels, and save your panel arrangements in a [layout](#)<sup>[42]</sup>.

## Available Panels

### How to access the list of available panels

- Select the Home tab > in the Windows section, select Panels:



The panels available in PowerShell Studio:

 Call Stack	Ctrl+Alt+P, K
 Console	Ctrl+Alt+P, C
 Debug Console	Ctrl+Alt+P, D
 Find Results	Ctrl+Alt+P, R
 Function Explorer	Ctrl+Alt+P, F
 Help	Ctrl+Alt+P, H
 Object Browser	Ctrl+Alt+P, B
 Output	Ctrl+Alt+P, O
 Performance	Ctrl+Alt+P, M
 Project	Ctrl+Alt+P, J
 Properties	Ctrl+Alt+P, P
 Snippets	Ctrl+Alt+P, S
 Toolbox	Ctrl+Alt+P, T
 Tools Output	Ctrl+Alt+P, L
 Variables	Ctrl+Alt+P, V
 Watch	Ctrl+Alt+P, W

For details about the content of each panel, see [Panels](#).

### 4.2.3.2 Working with Panels

This section shows how to work with panels by providing an overview of the most common tool panel tasks.

#### Opening a Panel

##### To open or re-open a panel

- In the Panels drop-down (**Home** tab > **Windows** section > **Panels**), select the panel to launch.

-OR-

- Execute the chorded keyboard shortcut for the panel you want to access: Simply press **Ctrl+Alt+P**, release, then press the corresponding character key of the chord:

	<u>Call Stack</u>	Ctrl+Alt+P, K
	<u>Console</u>	Ctrl+Alt+P, C
	<u>Debug Console</u>	Ctrl+Alt+P, D
	<u>Find Results</u>	Ctrl+Alt+P, R
	<u>Function Explorer</u>	Ctrl+Alt+P, F
	<u>Help</u>	Ctrl+Alt+P, H
	<u>Object Browser</u>	Ctrl+Alt+P, B
	<u>Output</u>	Ctrl+Alt+P, O
	<u>Performance</u>	Ctrl+Alt+P, M
	<u>Project</u>	Ctrl+Alt+P, J
	<u>Properties</u>	Ctrl+Alt+P, P
	<u>Snippets</u>	Ctrl+Alt+P, S
	<u>Toolbox</u>	Ctrl+Alt+P, T
	<u>Tools Output</u>	Ctrl+Alt+P, L
	<u>Variables</u>	Ctrl+Alt+P, V
	<u>Watch</u>	Ctrl+Alt+P, W

## Panel Buttons

Once a panel is displayed it can be adjusted by using the buttons at the top-right of the panel:



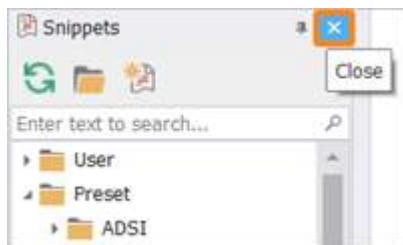
### From left to right

-  **Maximize / Restore** - This option is visible when there is more than one panel in a location. You can Maximize the panel to full-size in the location, or Restore the panel so that all panels in the location are visible.
-  **Auto Hide / Dock** - Auto Hide will vertically collapse, or "Hide" the panel in the location. Dock will open the panel to a fixed position in the location.
-  **Close** - Remove the panel from the location.

## Closing a Panel

### To close a panel

- In the top-right of the panel, click **Close**:



## Showing a Hidden Panel

### To show a hidden panel

*Hidden* panels are open, but are not docked or floating. They open when you click them, stay open while you use them, and hide when you click away.

To show a hidden panel, at the edge of the window (left, right, or bottom) click the panel tab that you want to show:



The panel will open and stay open until you click away.

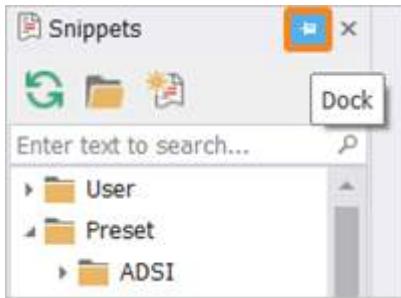
- You can see the tabs of hidden panels at the left, right, and bottom edges of the window.
- To keep a hidden panel open, dock it.

## Docking a Hidden Panel

### To dock a hidden panel

This action causes an open, hidden panel to "dock" in a fixed position.

- In the top-right of the panel, click the **pin** (to "pin" or "dock" the panel):



-OR-

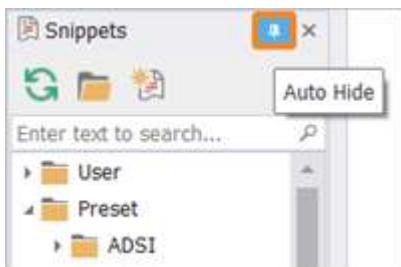
- Right-click the panel title bar and select **Dock**.

## Hiding a Panel

### To hide a panel

This action keeps a panel open, and causes it to "hide" when you click away from the panel.

- In the top-right of the panel, click the pin (to "unpin" or "auto hide"):



-OR-

- Right-click the panel title bar and select **Auto Hide**.

## Floating a Panel

### To float a panel

This action releases a panel from a docked position.

1. Right-click the panel title bar (or panel tab if the panel is grouped), and select **Float**.
2. Avoiding the docking indicator, drag and release the panel in the desired location.

-OR-

1. Drag the panel title bar (or panel tab if the panel is grouped). If the title bar contains a group of panels, dragging the title bar will drag the entire group.

2. Avoiding the docking indicator, release the panel in the desired location.

## Re-docking a Panel

---

### To re-dock a panel

This action returns a floating panel to its most recently docked position.

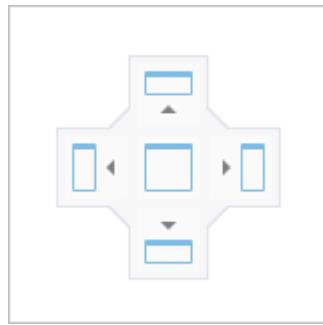
- Right-click the panel title bar and select **Dock**.

## Moving a Panel

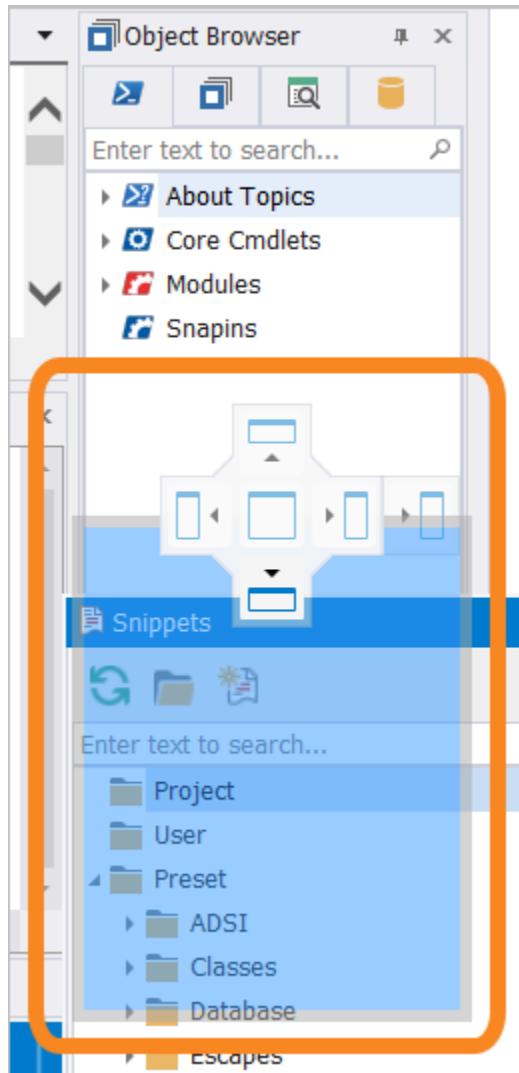
---

### To move a panel

You can left-click on the title bar of a panel and drag it to another location. The docking indicator will appear as soon as you begin to drag the panel:



You can drop the panel onto a docking hint to have it snap to the desired location, or leave the panel floating in the window. As shown by the blue highlighting in the example below, the Snippets panel will be docked below the Object Browser:

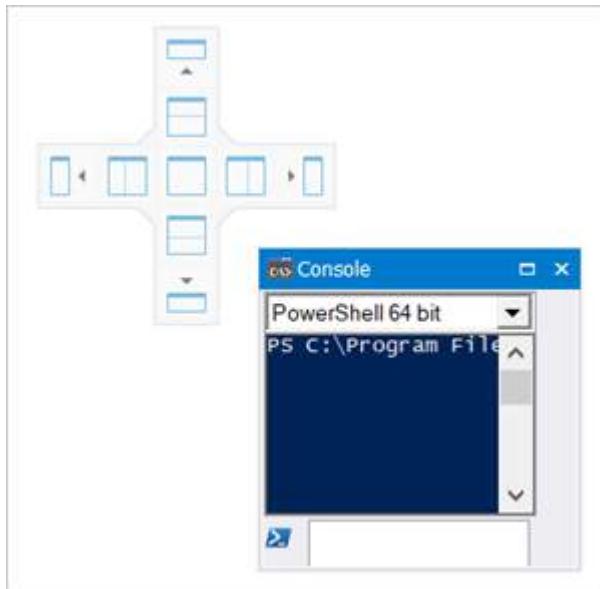


Another example of how to move a panel:

1. Drag the panel title bar toward the desired location. If the title bar contains a group of panels, dragging the title bar will move the entire group.

To separate a panel from a group, right-click the panel tab and click **Float**.

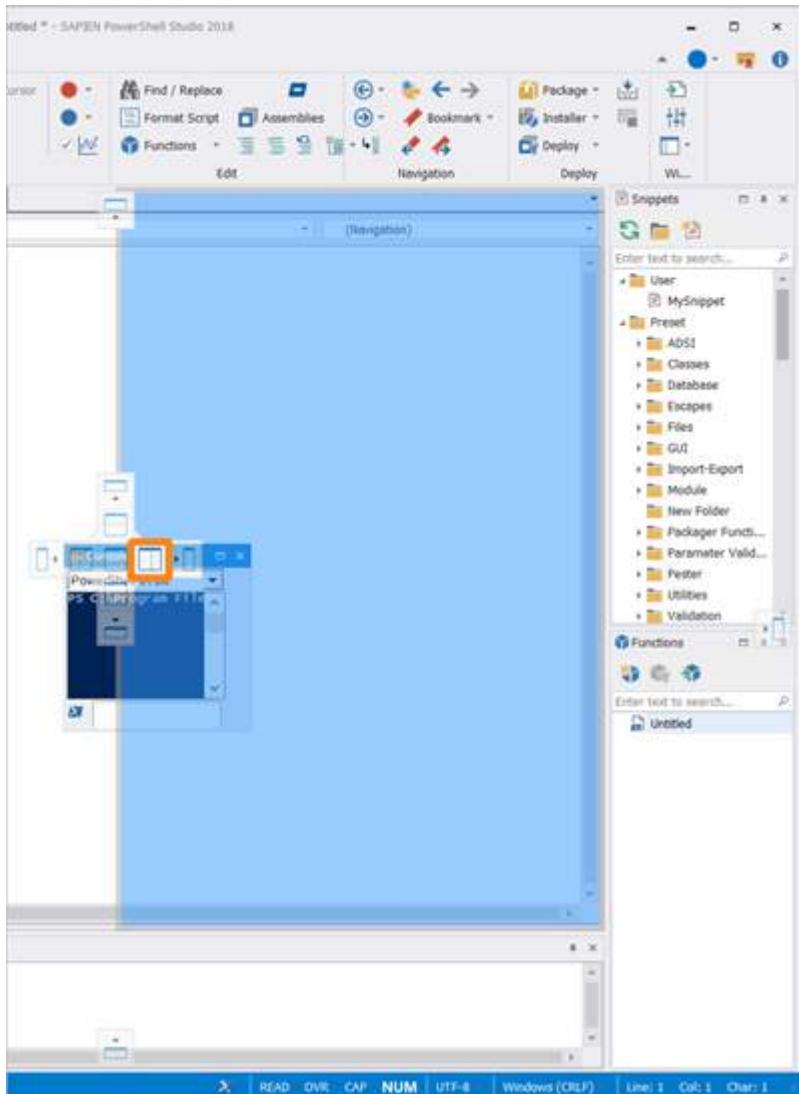
The docking indicator appears. The locations on the docking indicator represent locations in the PowerShell Studio layout:



For example, the center position corresponds to the center of the layout, the center-left position indicates the center-left position of the layout, and the left position indicates the left side of the layout.

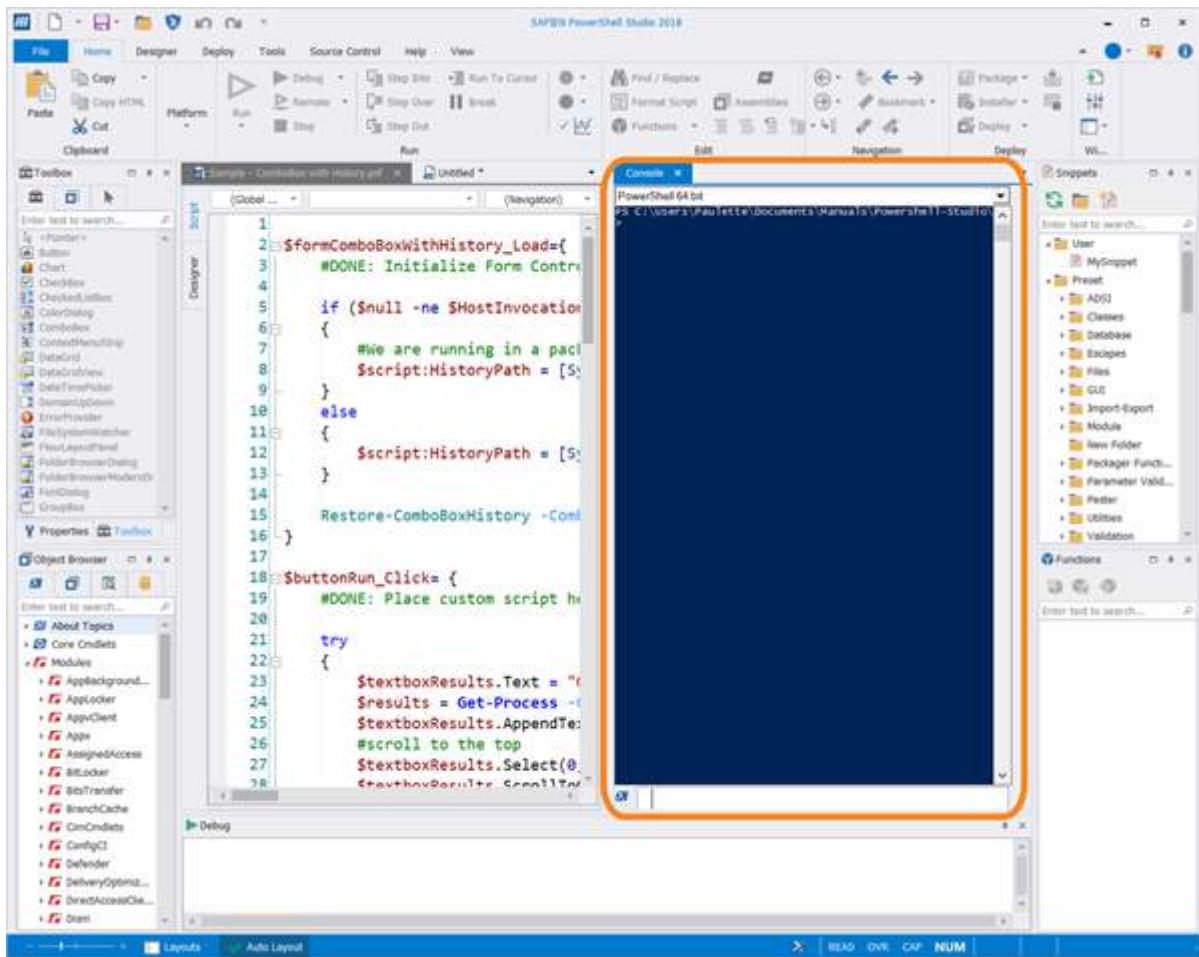
2. Drag the panel to a location on the docking indicator, and then drop it.

As you move the panel toward a position on the docking indicator, the corresponding part of the layout displays a blue overlay where the panel will dock:



When you drop the panel, it occupies the blue space:

# Basic Orientation



If you are having trouble dragging a tab or panel, dock it and try again.

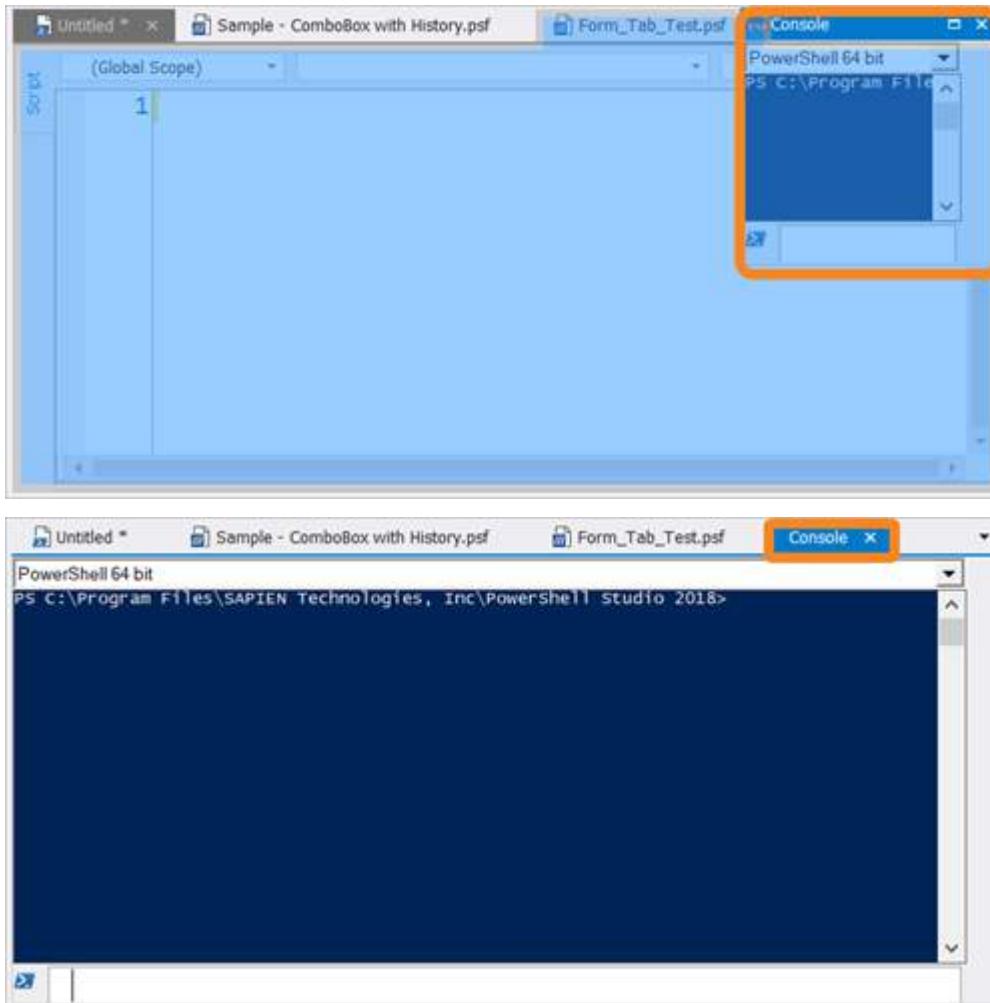
## Converting a Panel to a Tab

### To convert a panel to a tab

You can convert a panel to a tab of the PowerShell Studio *Script Editor* or *Designer* window.

- Drag the panel and drop it on the tab bar:

# Basic Orientation



-OR-

- Right-click the panel title bar (or panel tab if the panel is grouped), and then click **Dock as Tabbed Document**.

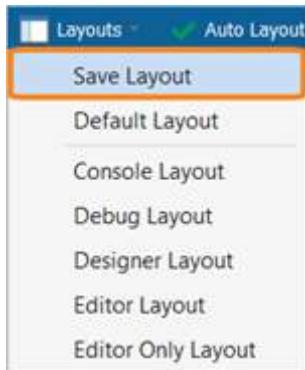
## Saving a Layout

### To save a new layout

A *layout* defines the position and visibility of the various PowerShell Studio panels.

To save your own custom layout:

1. Arrange the panels to your preferred layout.
2. From the Layouts menu on the status bar, click **Layouts > Save Layout**.



### 4.2.3.3 Layouts

A *layout* defines the position and visibility of the various PowerShell Studio panels. PowerShell Studio will automatically choose a layout based on what you are doing (via context sensitive Auto Layout). You can also choose from preset layouts, save your own custom layouts, and configure layout options.

## Preset Layouts

There are preset layouts available that configure PowerShell Studio for specific activities. To switch to a preset layout, click **Layouts** on the bottom-left of the status bar and select a layout from the list:



- **Default Layout**

The Default Layout is the same as the Editor layout.

- **Console Layout**

The Console Layout emphasizes the script editor and console panel.

- **Debug Layout**

The Debug Layout emphasizes the script editor, watch and output panels.

- **Designer Layout**

The Designer Layout emphasizes the code editor, toolbox and properties windows. This is ideal for developing PowerShell forms applications.

- **Editor Layout**

The Editor Layout emphasizes the code editor, project browser, object browser, output window, function explorer and snippet panels. This is a good general purpose panel layout.

- **Editor Only Layout**

The Editor Only Layout hides everything except the editor window.

## Auto Layout

On the bottom-left of the status bar, to the right of the Layouts menu, is the Auto Layout option which is used to enable or disable automatic context sensitive layout changes.

### To toggle Auto Layout on or off

Click the **Auto Layout** button:

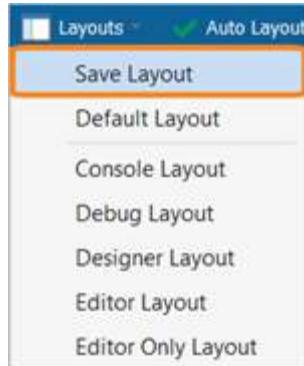


## Custom Layouts

You can save your own customized layout and access it from the Layouts menu on the status bar.

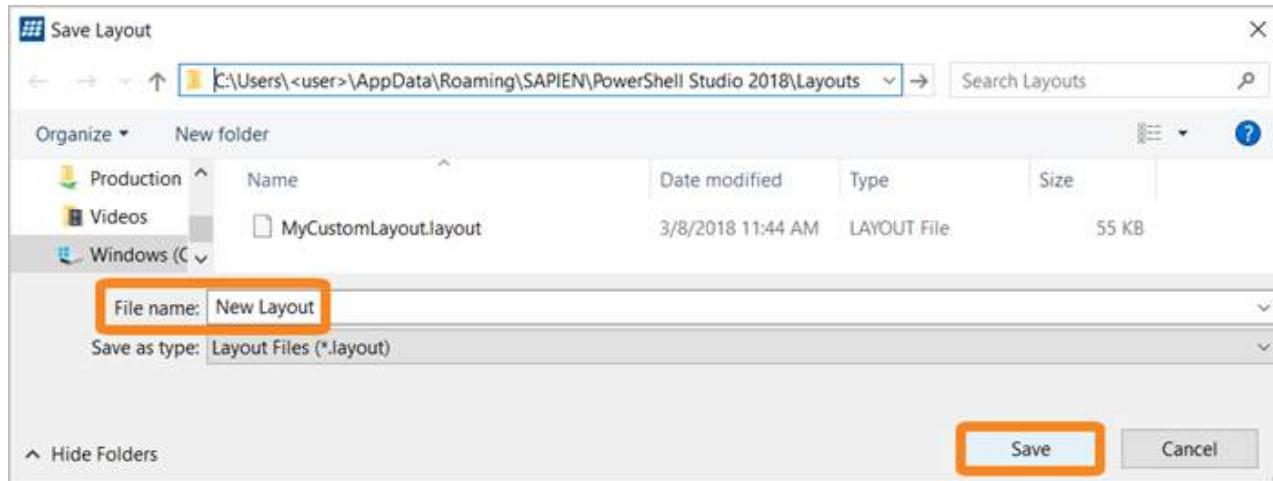
### To save a customized layout

Configure the panels in PowerShell Studio to your preferred layout and then click **Layouts > Save Layout**:



PowerShell Studio will display the Save Layout dialog. Type a **File Name** for your new layout, then click **Save**:

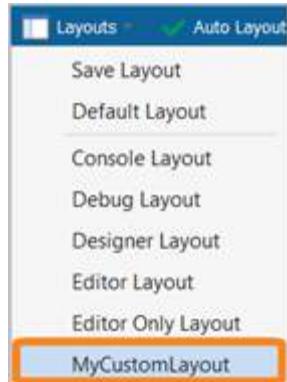
# Basic Orientation



Each layout pertains to a .layout file, which contains the state information of the panels in PowerShell Studio.

Layout files are saved in the following location: %Users%\<user>\AppData\Roaming\SAPIEN\PowerShell Studio <year>\Layouts

The new layout will appear in the list of available layouts:

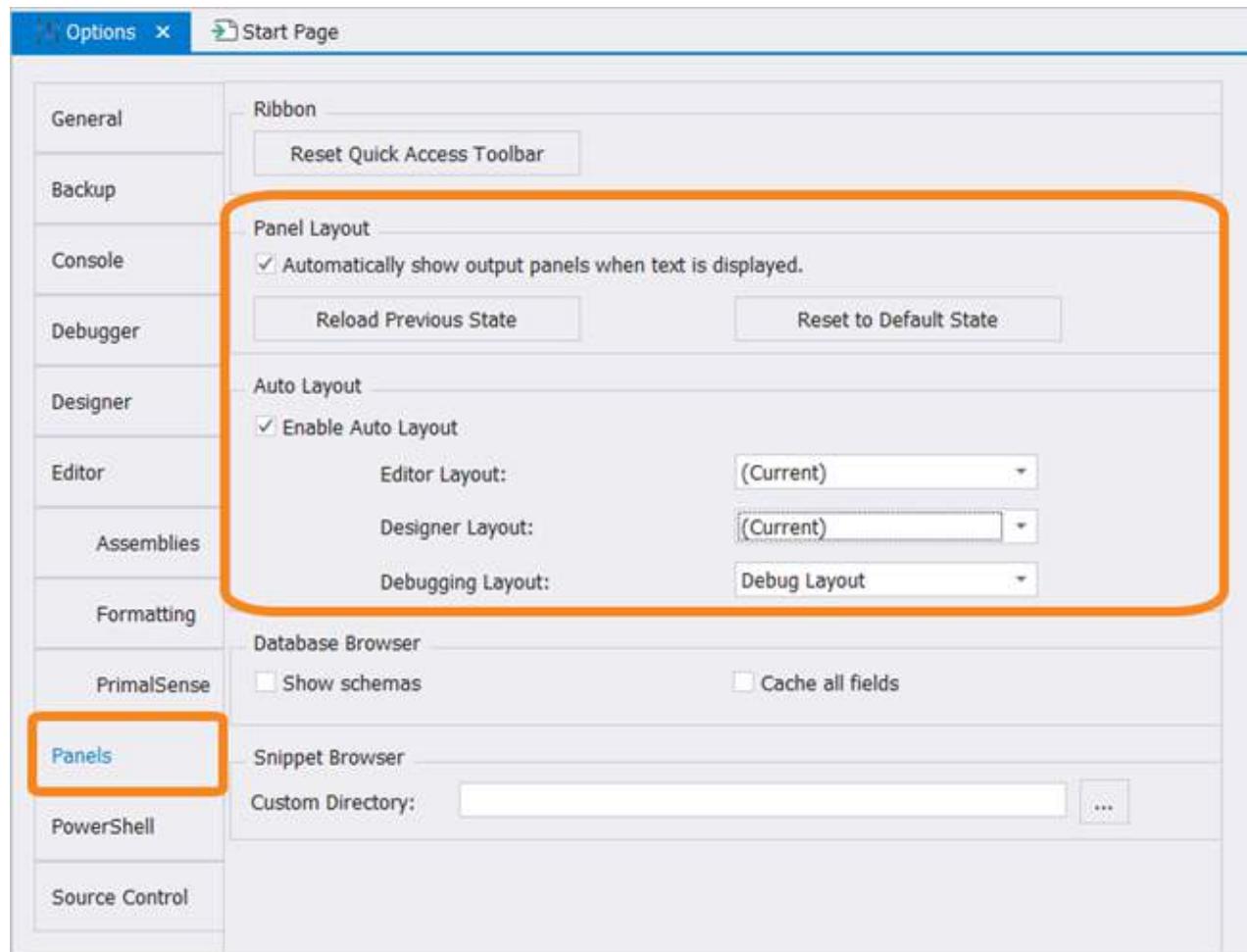


To maintain the current layout, disable **Auto Layout** on the status bar:



## Layout Options

Layout options can be configured on the Panels tab of the Options dialog (File > Options > Panels or Home > Options > Panels):



- i** Selecting **(Current)** for any Auto Layout option will keep the layout that you are currently using. For example, if the Editor Layout is set to **(Current)**, when you open the script editor a layout change will not be triggered.

## 5 Script Editor

PowerShell Studio is much more than just a script editor—it is a complete *environment* which includes dozens of built-in tools and functions to make scripting more efficient. At the heart of PowerShell Studio is the industries most powerful and flexible code editor. While it's easy to start using PowerShell Studio's editor without any training, some of its features can be easily overlooked. In this section you will learn about the script editor features, including tips for using PowerShell Studio more efficiently.

### 5.1 Editing Aids

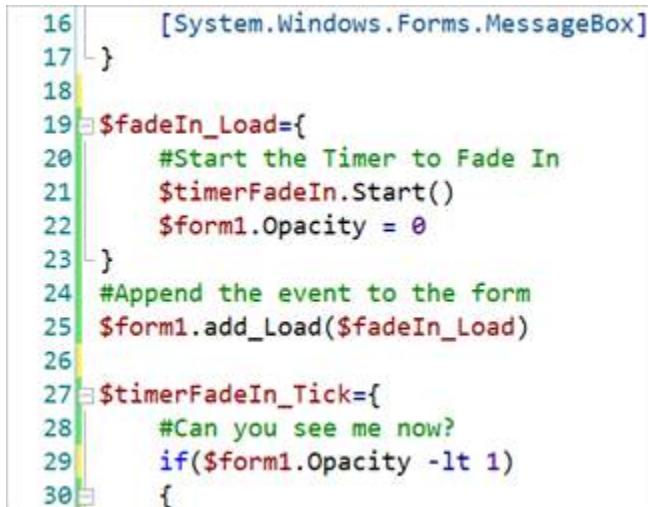
PowerShell Studio includes many features that make script editing easier.

#### 5.1.1 Line Numbering and Visual Features

PowerShell Studio uses line numbering and visual cues, such as coloring, to make your script editing easier.

#### Line Numbering and Color Status

Line numbers are displayed on the left edge of the editor panel:



The screenshot shows a code editor window with line numbers from 16 to 30 on the left. To the right of each line number is a colored vertical bar indicating the edit status: green for edited and saved lines, yellow for edited but unsaved lines, and no color for lines that have not been edited. The code itself is written in PowerShell, involving the System.Windows.Forms.MessageBox class and timer events.

```
16 [System.Windows.Forms.MessageBox]
17 }
18
19 $fadeIn_Load={
20     #Start the Timer to Fade In
21     $timerFadeIn.Start()
22     $form1.Opacity = 0
23 }
24 #Append the event to the form
25 $form1.add_Load($fadeIn_Load)
26
27 $timerFadeIn_Tick={
28     #Can you see me now?
29     if($form1.Opacity -lt 1)
30 }
```

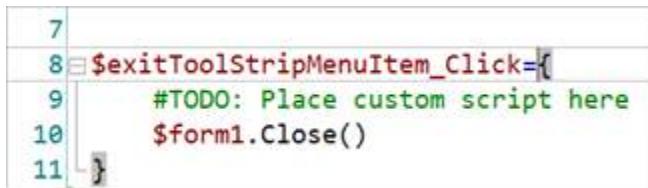
The colored bar to the right of the line numbers indicates the edit status of your code:

- **Green**  
Lines marked with green have been edited and saved since you opened the file.
- **Yellow**  
Lines marked with yellow have been edited but not yet saved.
- **No Color**  
Lines without a color have not been edited in this session.

## Pair Highlighting

PowerShell Studio also highlights pairs of braces, brackets, and parentheses in grey: {}, [], and () .

For example, when you click on a bracket its partner will be highlighted:



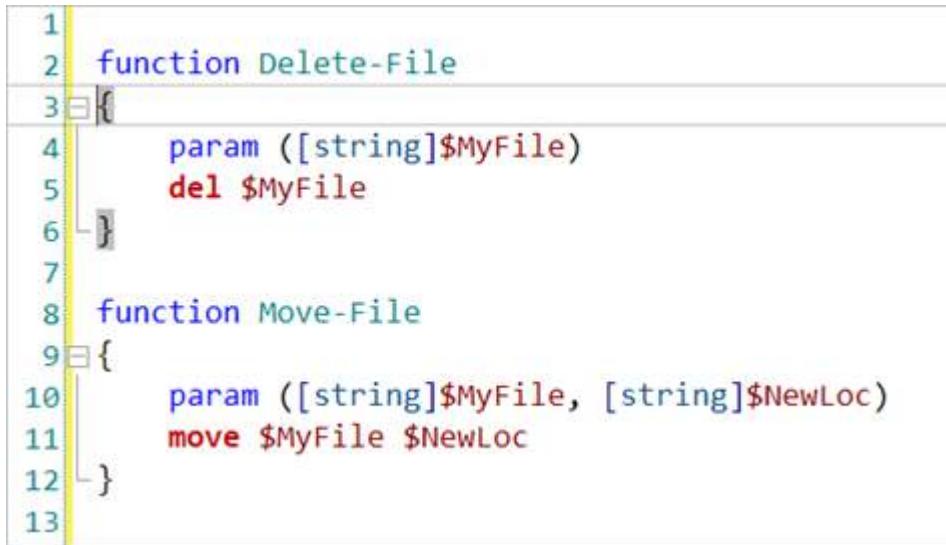
```
7
8 $exitToolStripMenuItem_Click={
9     #TODO: Place custom script here
10    $form1.Close()
11 }
```

## 5.1.2 Code Folding

Code folding is a feature that allows you to collapse or expand sections of your code (regions). Collapsing sections of code in a long script makes it easier to focus on the section you're working on. PowerShell Studio automatically creates regions, and you can also create your own.

## Automatic Regions

PowerShell Studio automatically creates foldable regions from declared functions, multi-line comment blocks, and script blocks:



```
1
2 function Delete-File
3 {
4     param ([string]$MyFile)
5     del $MyFile
6 }
7
8 function Move-File
9 {
10    param ([string]$MyFile, [string]$NewLoc)
11    move $MyFile $NewLoc
12 }
13
```

When collapsed, PowerShell Studio properly displays line numbering, accounting for the lines contained within the collapsed (or folded) region. This ensures that line number-based error messages and other information remain accurate:

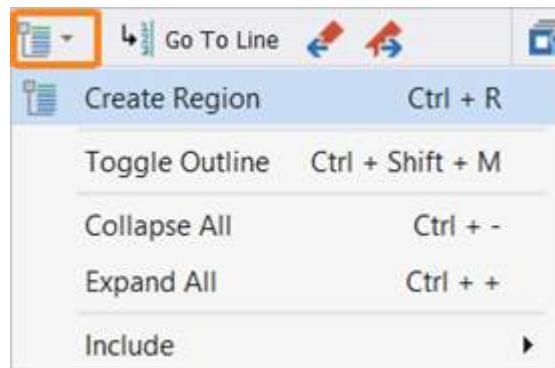
```
1
2 function Delete-File
3 { ...
4
5 function Move-File
6 {
7     param ([string]$MyFile, [string]$NewLoc)
8     move $MyFile $NewLoc
9 }
10
11 }
```

## Named Regions

A named region is a region that remains persistent even when the file is opened and closed.

### To create a named region

Highlight a block of code, then select **Home > Edit > Regions > Create Region (Ctrl+R)**:



A region is created with the default name 'RegionName', which is highlighted and ready for you to edit:

```
1
2 #region RegionName
3 function Delete-File
4 {
5     param ([string]$MyFile)
6     del $MyFile
7 }
8
9 function Move-File
10 {
11     param ([string]$MyFile, [string]$NewLoc)
12     move $MyFile $NewLoc
13 }
14 #endregion RegionName
15
```

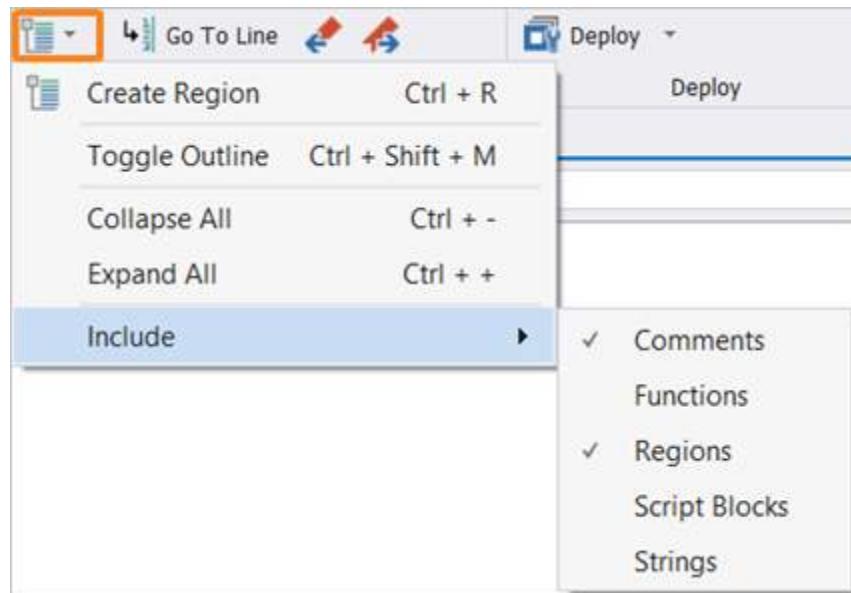
Named regions are convenient because the name remains visible when folded—indicating what the region contains:

```
1
2 #region FileFunctions
15
```

You can also create a named region manually by specifying the `#region` and `#endregion` keywords on comment lines within your script.

## Manipulating Regions

The options for working with outlines and regions are located in **Home > Edit > Regions**:



- **Create Region (Ctrl+R)**

Creates a persistent folding region.

- **Toggle Outline (Ctrl+Shift+M)**

Collapses or expands the current outline or region.

- **Collapse All (Ctrl+Minus Sign)**

Collapses all the expanded outlines and regions.

- **Expand All (Ctrl+Plus Sign)**

Expands all the collapsed outlines and regions.

- **Include (Comments, Functions, Regions, Script Blocks, Strings)**

Select which regions collapse and expand when you use the collapse / expand commands.

**i** When you load a script file, the editor will collapse all of the nodes selected under 'Include'. This option is set by default under Home > Options > Editor > Collapse regions on load.

### 5.1.3 Reference Highlighting

PowerShell Studio's *reference highlighting* feature makes it easy to highlight references in your code simply by double-clicking on a variable, identifier, member, function, or any other object.

Reference highlighting recognizes the different ways variables and parameters are referenced:

```
15 function Load-FolderIntoTree
16 {
17     param ( [System.Windows.Forms.TreeView]$TreeView,
18             [string]$Directory)
19
20     #Disable Drawing
21     $TreeView.BeginUpdate()
22
23     $TreeView.Nodes.Clear()
24     $folderName = [System.IO.Path]::GetFileName($Directory)
25     if ($folderName -eq '')
26     {
27         $folderName = $Directory
28     }
29     $TreeView.BeginUpdate()
```

👉 You can also use the **Ctrl+W(ord)** keyboard shortcut to select the current word and highlight the references.

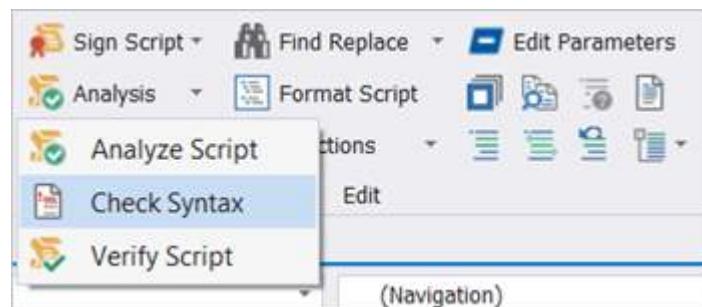
#### 5.1.4 Syntax Checking

PowerShell Studio constantly analyzes the code you type. Syntax errors are indicated by a red exclamation point (!) in the margin to the left of the line with the error in the code editor.

👉 Hover over the exclamation point to display a tool tip with information about the error:

```
13
14 $file = Get-Content .SPS.txt
15     [regex]$regex = "CREATE PROC(EDURE)?\s+"
! 16 $file | {
17
At line:16 char:9 Expressions are only allowed as the first element of a pipeline.
18
19
```

You can also manually invoke the syntax checker from the **Home tab > Edit section > Analysis menu > Check Syntax**:

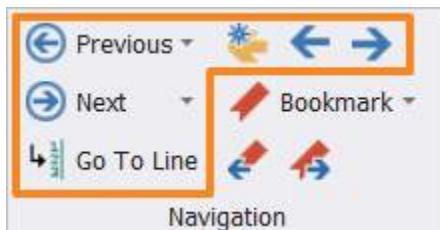


## 5.2 Navigation and Bookmarks

The Navigation section of the Home tab provides options for [navigating](#)<sup>[52]</sup> and [bookmarking](#)<sup>[53]</sup>, enabling you to quickly move between different locations in your scripts:



### Navigating



- **Previous**

Previous	Previous Function	Ctrl + Shift + F12
Previous Change		Ctrl + Shift + Up
Previous Occurrence	Ctrl + Shift + Alt + Up	

- **Previous Function** (*Ctrl+Shift+F12*)

Go to the previous function declaration.

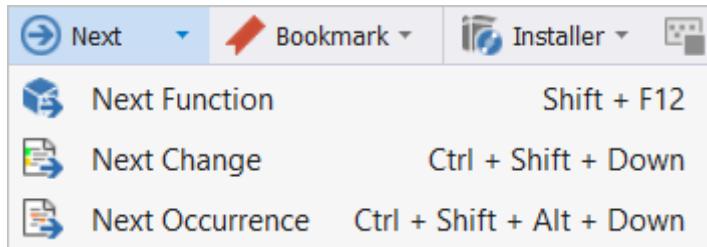
- **Previous Change** (*Ctrl+Shift+Up*)

Go to the previous position where you made an edit (i.e., where you typed or deleted something.)

- **Previous Occurrence** (*Ctrl+Shift+Alt+Up*)

Go to the previous occurrence of the selected item.

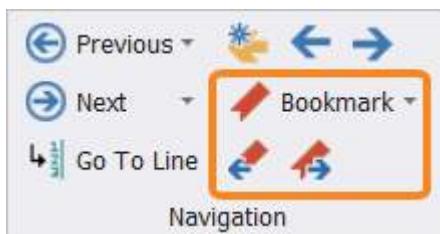
- **Next**



- **Next Function (Shift+F12)**  
Go to the next function declaration.
- **Next Change (Ctrl+Shift+Down)**  
Go to the next position where you made an edit (i.e., where you typed or deleted something.)
- **Next Occurrence (Ctrl+Shift+Alt+Down)**  
Go to the next occurrence of the selected item.
- **Go To Line (Ctrl+G)**  
Go to a specific line number.
- **Last Edit (Ctrl+E)**  
Go to the last position where there was a modification.
- **Navigate Backward (Ctrl+Shift+Minus Sign)**  
Navigate backward to the previous location.
- **Navigate Forward (Ctrl+Shift+Plus Sign)**  
Navigate forward to the previous location.

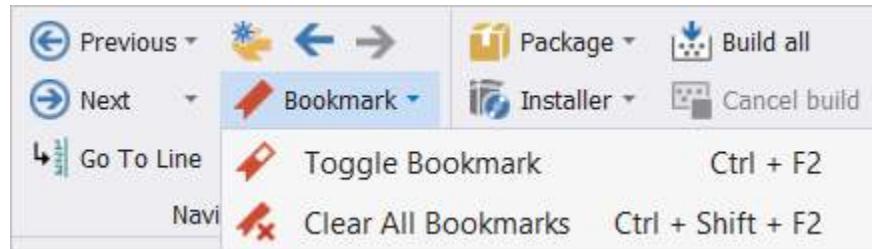
## Bookmarking

Use bookmarks in your files to mark locations and navigate between them:



### To toggle a bookmark on or off

Position the caret on the line where you want to add or remove a bookmark, or on a line that is already bookmarked, then select **Bookmark > Toggle Bookmark (Ctrl+F2)**.



Bookmarks are displayed to the left of the line numbers in the script editor window, as shown below:

```
32 [adsi]$user="WinNT://$computer/$account,user"
33 #get current password age
34 [int]$oldage=$user.passwordage[0]
35 if ($user.name) {
36     $user.SetPassword($password)
37     sleep -milliseconds 500
38 }
39 else {
40     $msg="Failed to get $account on "+$computer.ToUpper()
41     Write-Warning $msg
42     return
43 }
```

👉 To remove all bookmarks in the active document, select **Bookmark > Clear All Bookmarks** (*Ctrl+Shift+F2*).

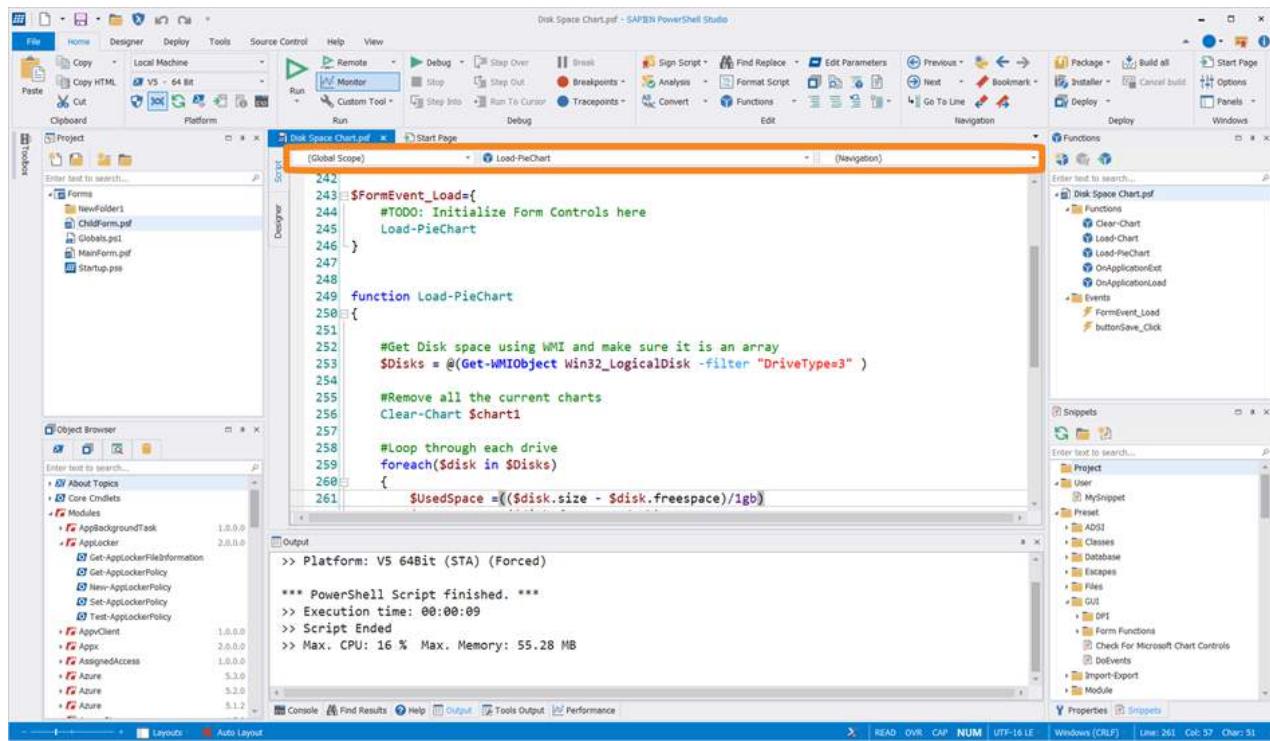
## To move between bookmarks

On the **Home** tab > click **Previous Bookmark** (*Shift+F2*) or **Next Bookmark** (*F2*) to quickly jump between bookmarks:



### 5.2.1 Navigation Bar

Above the editor is a navigation bar which allows you to jump between functions, workflows, events, and more:



## Navigation Bar - Options

The navigation bar contains three drop-downs.

From left to right:

- Global Scope**

Jump to class or enumerator declarations.

- Function**

Jump to a specific function, event, workflow, class, or configuration.

- Navigation**

Jump to the caret position, last edited position, current debug position, breakpoints, tracepoints, bookmarks, syntax errors, or comments (#MARK; #TODO).

### Global Scope

The Global Scope drop-down lets you jump to class or enumerator declarations contained within the file:

The screenshot shows two separate code editors within the PowerShell Studio interface. The top editor displays a script named 'WineGlass\_1.ps1' with the following code:

```

150
151     #Constructors
152     WineGlass () { }
153
154     WineGlass ([Wine]$Wine, [int]$Size, [int]$Pour)
155

```

A context menu is open at the end of the line '150'. The menu items visible are 'Properties', '\$Consumed', and '\$TotalPoured'. The 'Properties' item is highlighted.

The bottom editor displays a script named 'WineSweetness' with the following code:

```

13
14
15 enum WineSweetness
16 {
17     VeryDry
18     Dry
19     Medium
20     Sweet

```

A context menu is open at the start of the 'enum' keyword '15 enum'. The menu items visible are '<#...', 'FormEvent\_Load', 'buttonSave\_Click', 'Clear-Chart', 'Load-Chart', and 'Load-PieChart'. The 'Load-Chart' item is highlighted.

## Function

Use the Function drop-down to jump to a specific function, event, workflow, class, or configuration:

The screenshot shows two separate code editors within the PowerShell Studio interface. The top editor displays a script named 'Disk Space Chart.ps1' with the following code:

```

207     {
208         $ChartCont
209     }
210
211 #endregion
212
213

```

A context menu is open at the start of the 'function' keyword '207 function'. The menu items visible are 'FormEvent\_Load', 'buttonSave\_Click', 'Clear-Chart', 'Load-Chart', and 'Load-PieChart'. The 'Load-Chart' item is highlighted.

The bottom editor displays a script named 'Disk Space Chart.ps1' with the following code:

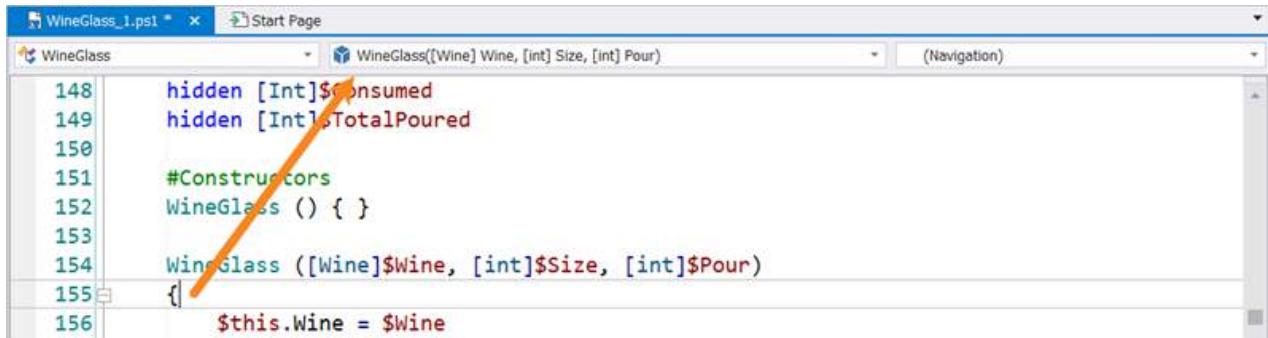
```

1 #-----
2
3 #region Chart Helper Functions
4 <#...
39
40 function Load-Chart
41 {
42     Param( #$XPoints, $YPoints, $XTitle, $YTitle, $Title, $ChartStyle)
43     [Parameter(Position=1,Mandatory=$true)]

```

A context menu is open at the start of the 'function' keyword '40 function'. The menu items visible are '<#...', 'Load-Chart', and 'Load-PieChart'. The 'Load-Chart' item is highlighted.

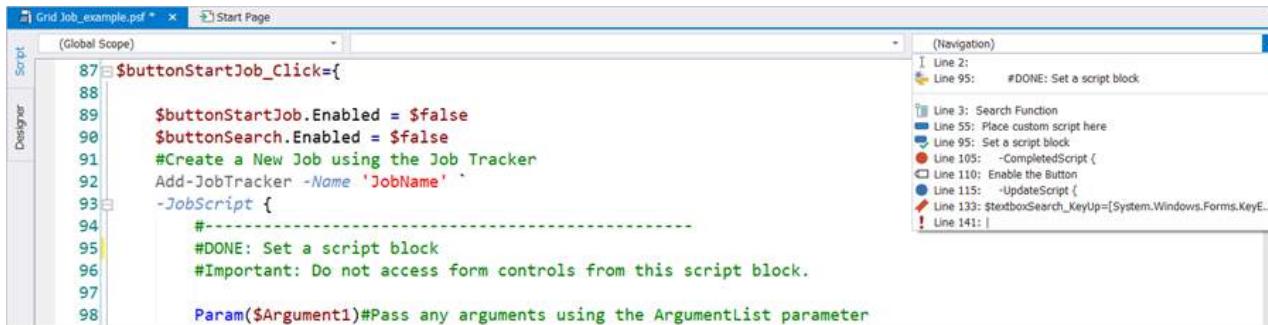
The Function drop-down reflects the context of the caret position as you navigate within the script. For example, when the caret is within the scope of a class, the Function drop-down reflects the class' member declaration:



```
WineGlass_1.ps1 * Start Page
WineGlass
148     hidden [Int]$Consumed
149     hidden [Int]$TotalPoured
150
151     #Constructors
152     WineGlass () { }
153
154     WineGlass ([Wine]$Wine, [int]$Size, [int]$Pour)
155     {
156         $this.Wine = $Wine
```

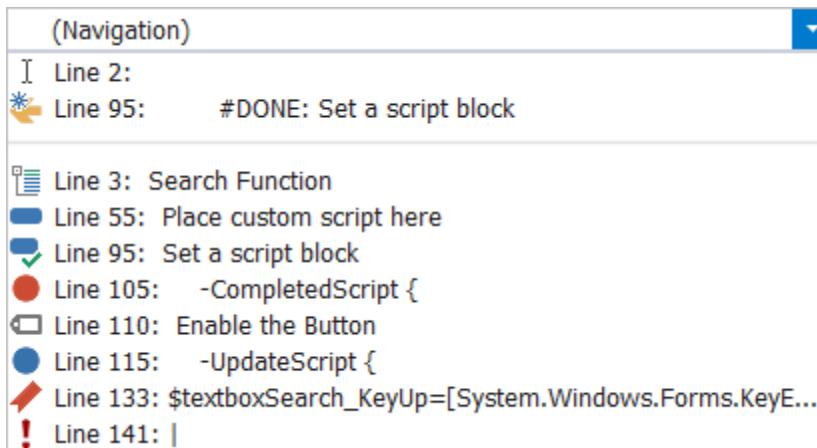
## Navigation

The Navigation drop-down allows you to jump to specific positions that include the caret position, the last edited position, current debug position, breakpoints, tracepoints, bookmarks, syntax errors, and more:



```
Grid Job_example.ps1 * Start Page
(Global Scope) (Navigation)
87 $buttonStartJob_Click={
88
89     $buttonStartJob.Enabled = $false
90     $buttonSearch.Enabled = $false
91     #Create a New Job using the Job Tracker
92     Add-JobTracker -Name 'JobName'
93     -JobScript {
94         #-----
95         #DONE: Set a script block
96         #Important: Do not access form controls from this script block.
97
98         Param($Argument1)#Pass any arguments using the ArgumentList parameter
```

The Navigation drop-down provides a preview of the line contents in order to help direct you to the correct position:



## TODO / DONE Task Comments

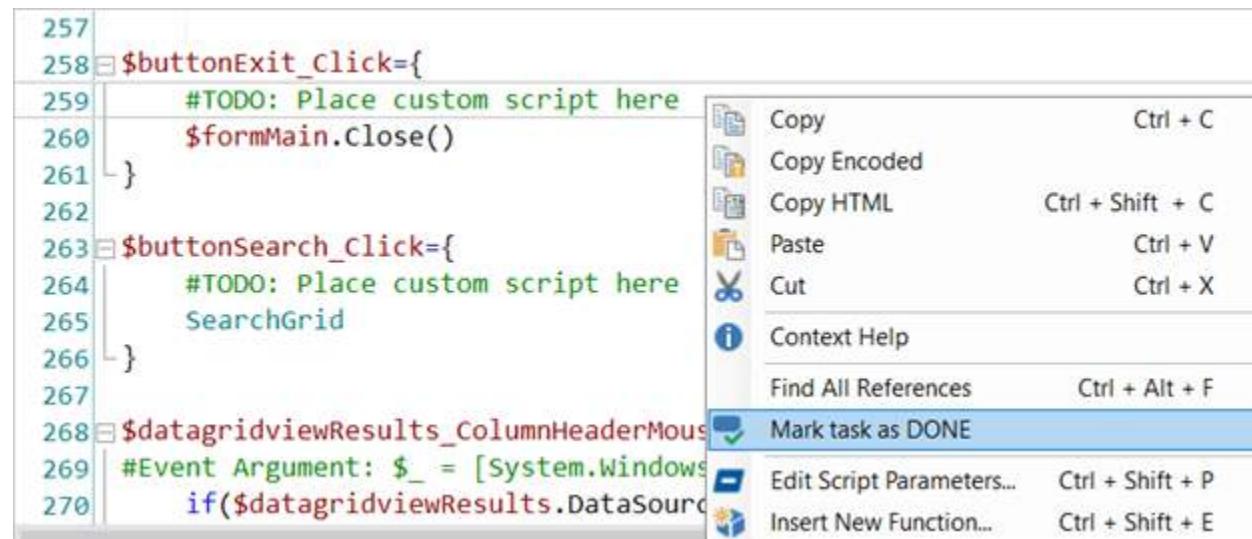
PowerShell Studio automatically includes TODO comments when you create event handlers for GUI applications. These comments are reminders that you need to place your custom script in the event script block:

```

257
258 $buttonExit_Click={
259     #TODO: Place custom script here
260     $formMain.Close()
261 }
262
263 $buttonSearch_Click={
264     #TODO: Place custom script here
265     SearchGrid
266 }
267

```

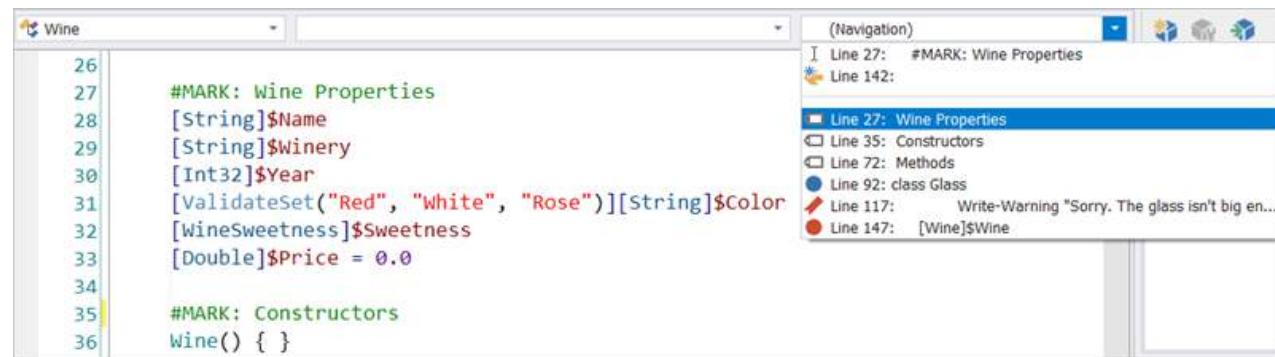
To mark a TODO comment as complete, right-click on the associated line in the script and select **Mark task as DONE**:



- 💡 DONE comments will appear in the Navigation menu with a green check mark as you complete the tasks.

### MARK Comments

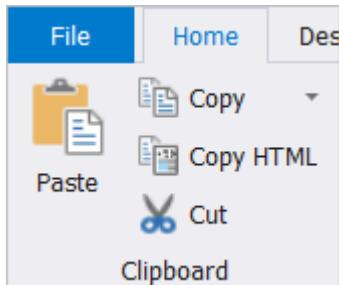
Comments starting with #MARK: will display in the Navigation menu:



- 💡 Use MARK comments to designate navigation points within your script

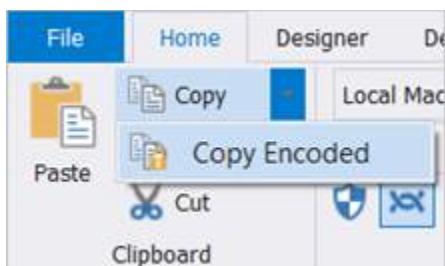
## 5.3 Clipboard Integration

PowerShell Studio provides the following Copy, Cut, and Paste clipboard functionality:



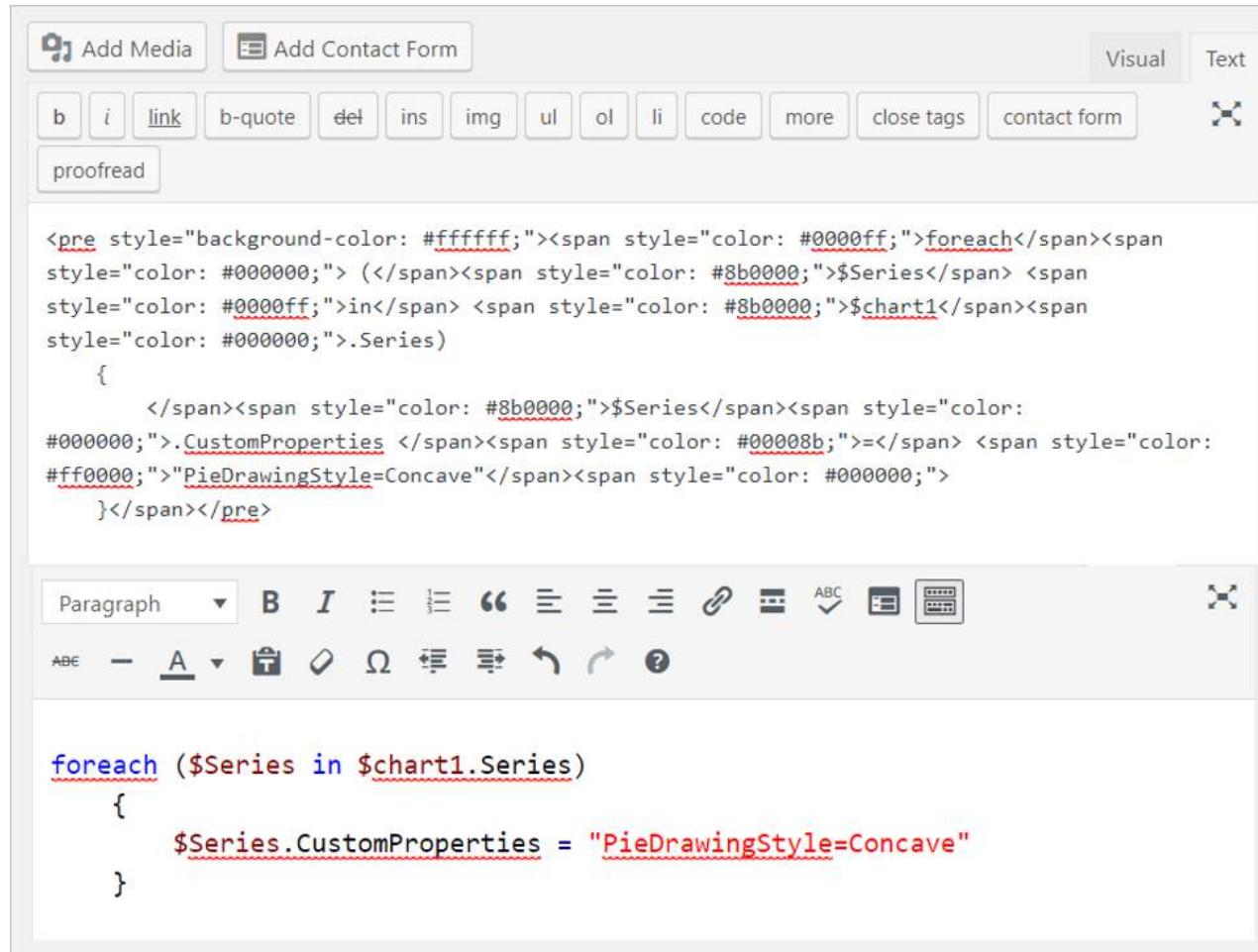
- **Copy (Ctrl+C)**  
Copy the selection and put it on the clipboard.
- **Copy Encoded**  
Copy the selection and put it on the clipboard as Base64 encoded text.
- **Copy HTML (Ctrl+Shift+C)**  
Copy the selection and put it on the clipboard as formatted HTML, for pasting into web applications.
- **Cut (Ctrl+X)**  
Cut the selection and put it on the clipboard.
- **Paste (Ctrl+V)**  
Paste the contents of the clipboard.

Use the **Copy Encoded** option from the **Copy** pull-down to Base64 encode the selected text and copy it to the clipboard:



```
14: 
15: Get-Process -Name 'PowerShell Studio'
16: RwB1AHQALQBQAHIAbwBjAGUAcwBzACAAALQBOAGEAbQB1ACAAJwBQAG8AdwB1AHIAUwBoAGUAbABsACAAUwB0AHUAZABpAG8AJwA=
17: 
```

 **Copy HTML** is an easy way to include code in a blog post. Simply select the code in PowerShell Studio and click **Home > Copy HTML** on the ribbon (or right-click the selected code in the editor and select **Copy HTML**), and then paste the HTML for the code in the Text view of the blog post. The result is perfectly copied code, including color and formatting.



## 5.4 Find and Replace Options

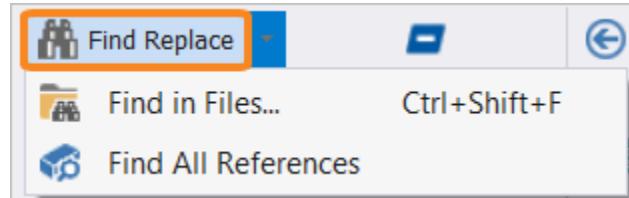
PowerShell Studio includes extensive, powerful search and replace features, allowing you to revise a file's contents quickly and easily.

### 5.4.1 Find and Replace

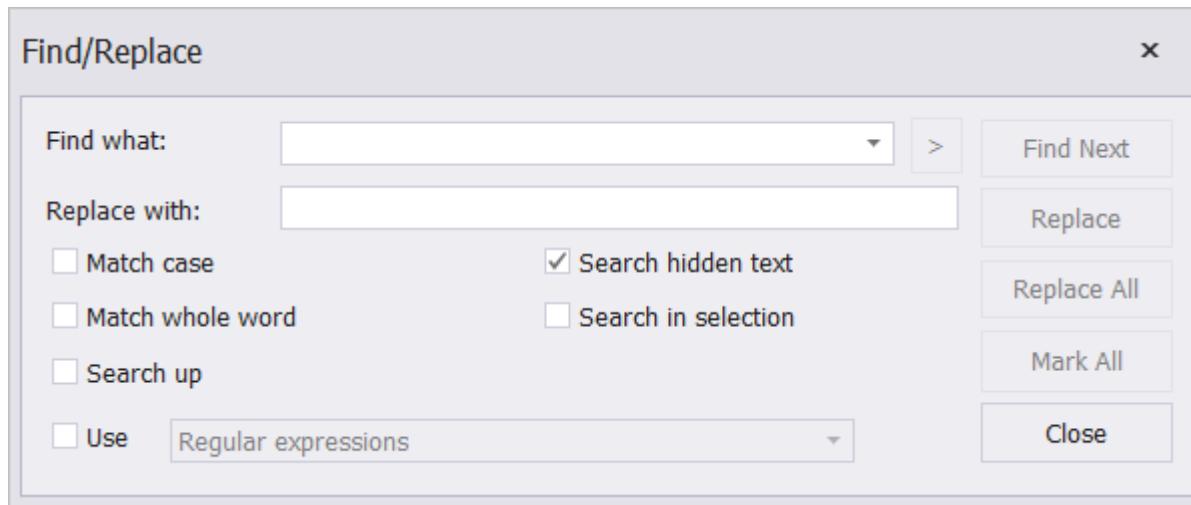
This topic explains how to search, and also how to find and replace.

#### How to use Find/Replace

Basic searching is easily accomplished from **Home** > in the Edit section, **Find Replace (Ctrl+F)**:



The Find/Replace dialog allows you to specify parameters for your search:



**i** The Find/Replace dialog will stay open while you edit a script.

The Find/Replace dialog options:

- **Find what**

Enter the text you are searching for. Use the drop-down list in the right of the field to access search term history.

- **Replace with**

The replacement text (optional).

- **Match case**

Makes the search case sensitive.

- **Match whole word**

Only finds occurrences of the search text that appear as whole words.

- **Search up**

Searches upwards from the cursor position.

- **Use**

Enables special processing of the search text. There are two options available:

- **Regular expressions**

Uses the full power of regular expressions in your search strings.

- **Wildcard matching**

Uses ? to represent a single character, or \* to represent multiple characters.

- **Search hidden text**

Includes folded code regions in the search.

- **Search in selection**

Confines the search to a highlighted section of code.

- **Find Next**

Find the next occurrence of your search string.

- **Replace**

Replace the search string occurrence with the replacement text.

- **Replace All**

Replace each occurrence of the search string with the replacement text.

- **Mark All**

Places a bookmark at each location where your search string occurs.

👉 Use *F2* and *Shift+F2* to move back and forth between bookmarks.

- **Close**

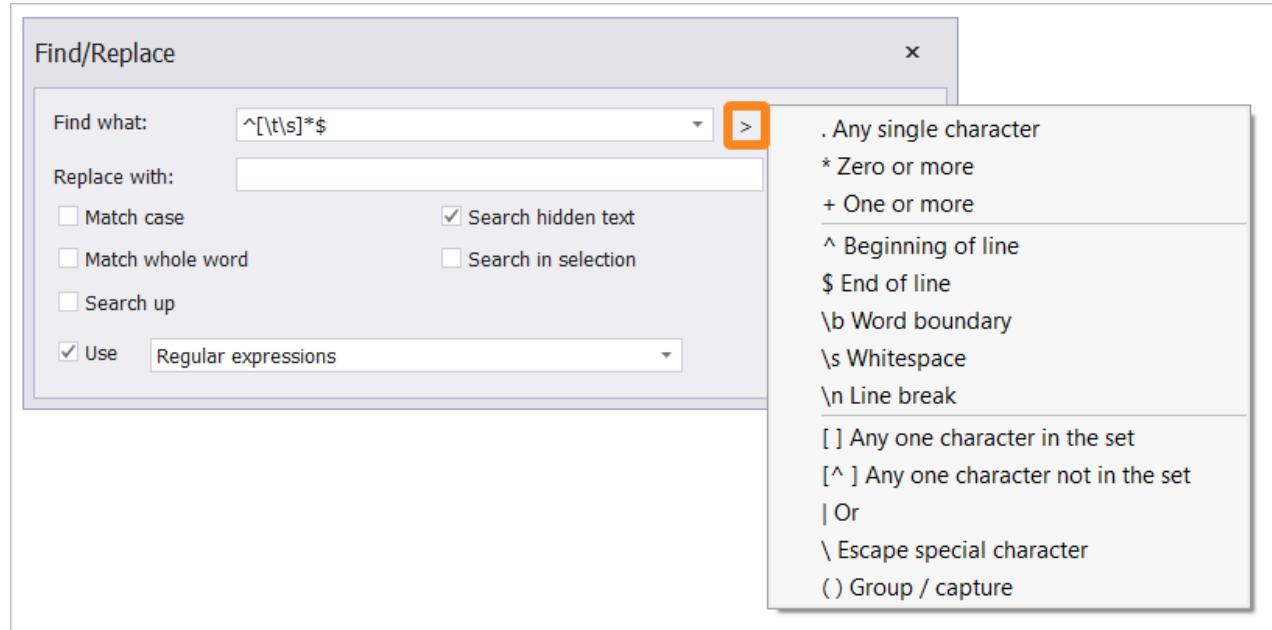
Close the Find/Replace dialog.

💡 Replace always operates on the entire script regardless of whether any code is highlighted or not, unless the 'Search in selection' option is checked.

## Regular Expression Searching

Checking the **Use** checkbox in the Find/Replace dialog enables a drop-down list where you can select **Regular expressions** or **Wildcards**.

👉 Selecting **Regular expressions** enables a button to the right of the **Find what:** field that displays a quick list of common regular expressions that you can use to quickly add components to your search text:

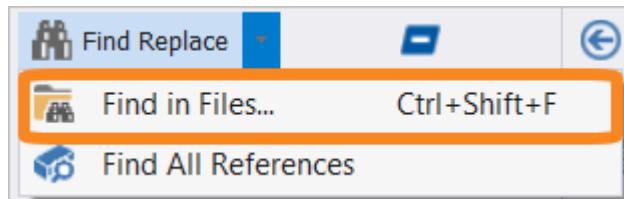


### 5.4.2 Find in Files

Find in Files allows you to search multiple files for specific text without having to open each file individually.

## How to use Find in Files

You can access the Find in Files tool from the **Home** tab > **Edit** section > **Find Replace** menu > **Find in Files...** (**Ctrl+Shift+F**):



Enter your search criteria in the Find in Files dialog, and then click **Find in Files**:



The Find in Files dialog options:

- **Find what**

Enter the text you are searching for. Use the drop-down list in the right of the field to access search term history.

- **Match case**

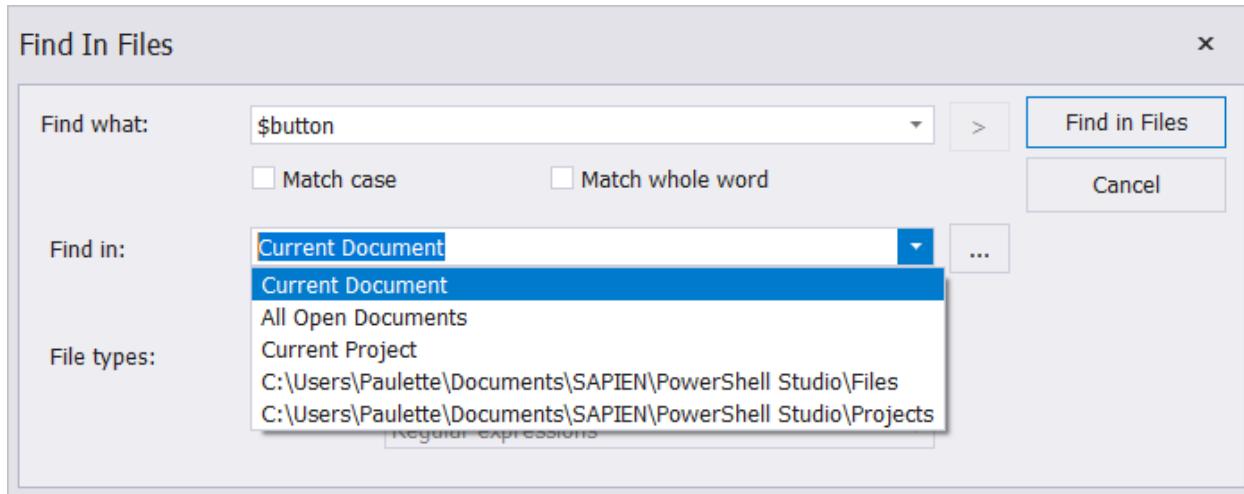
Make the search case sensitive.

- **Match whole word**

Find occurrences of the search text that appear as whole words.

- **Find In**

Designate the search folder by selecting from the drop-down list options. To select a different folder, use the More Options button (...) to the right of the **Find In** field.



- #### o Current Document

Search in the current document only.

- ### ○ All Open Documents

Search in all of the open documents.

- #### o Current Project

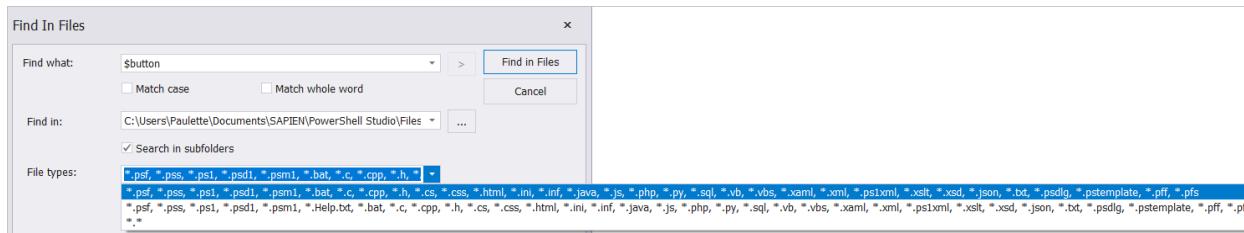
Search in all of the project files.

- Search in subfolders

Checked to search the subfolders recursively. If unchecked, the root directory will be searched.

- File Types

Specify the file types to be searched. By default, all PowerShell Studio file types are included.



- Use

Enables special processing of the search text. There are two options available:

- o Regular expressions

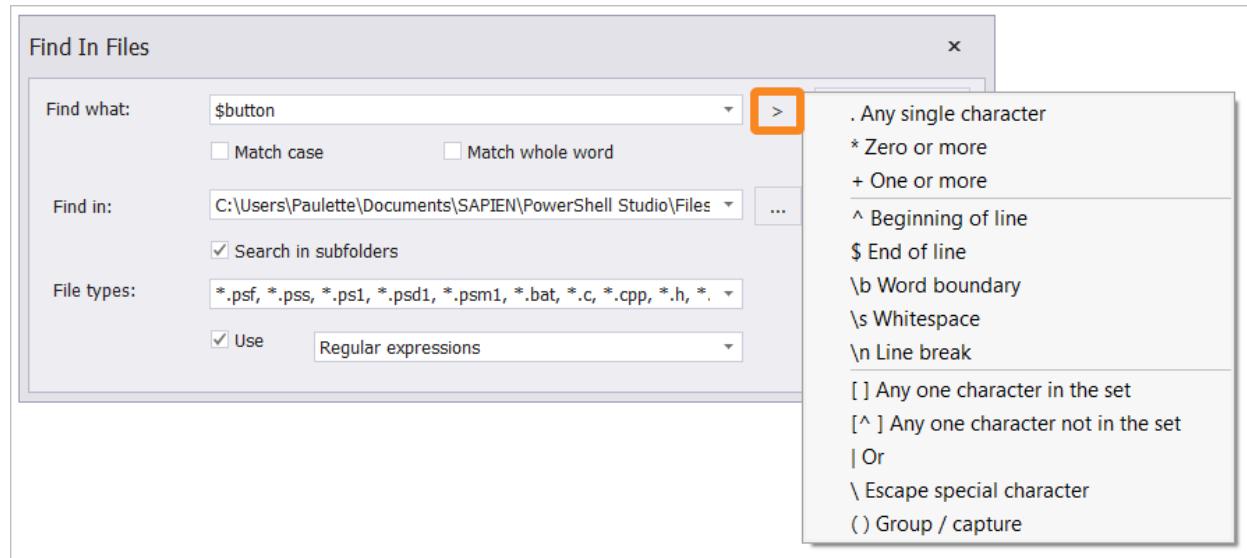
Uses the full power of regular expressions in your search strings.

- ### ○ Wildcard matching

Uses ? to represent a single character, or \* to represent multiple characters.

Checking the **Use** checkbox in the Find in Files dialog enables a drop-down list where you can select **Regular expressions** or **Wildcards**.

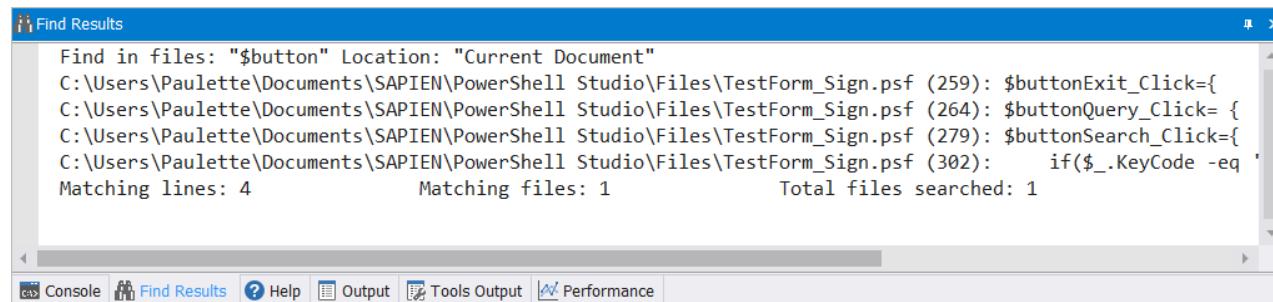
Selecting **Regular expressions** enables a button to the right of the **Find what:** field that displays a quick list of common regular expressions that you can use to quickly add components to your search text:



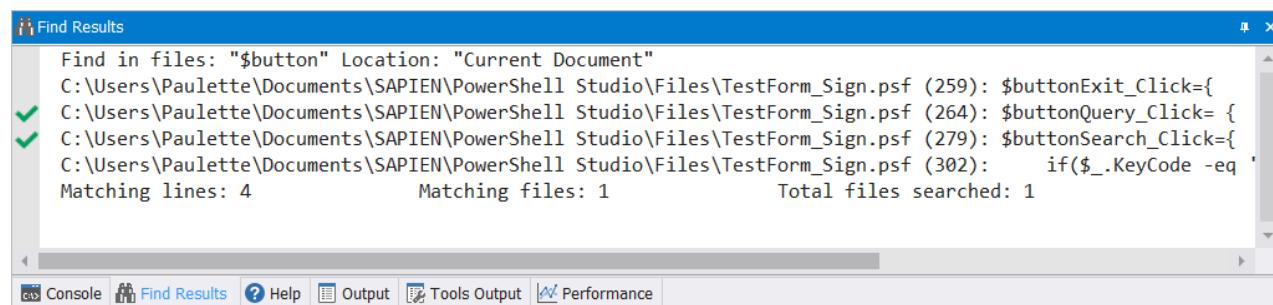
After you select the search options, click the Find in Files button to start the search.

## Search Results

The search results will appear in the [Find Results](#) (209) panel as the search progresses:

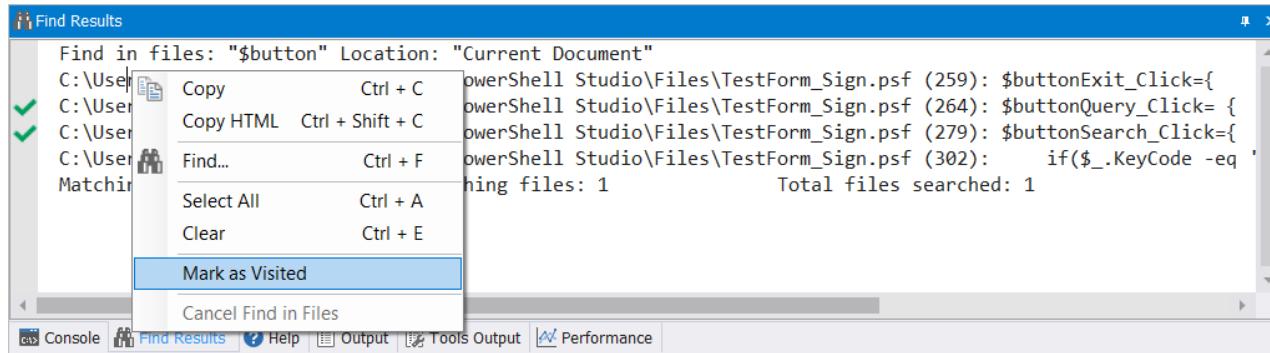


Double-clicking on a search result will open the file at the line specified in the result. Viewed results are indicated by a green check mark on the left column:



The check mark indicator helps you keep track of all the locations you visited.

To manually mark a result as visited or unvisited, right-click on a result and select **Mark as Visited** or **Mark as Unvisited**:

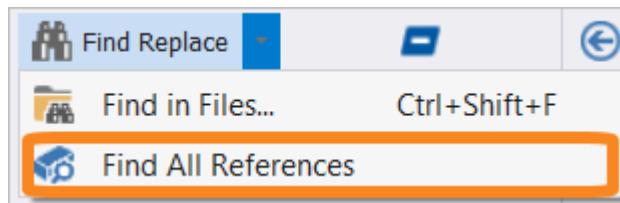


### 5.4.3 Find All References

Find All References builds upon PowerShell Studio's [reference highlighting](#) feature. In addition to highlighting all references of the object in the Editor, the references are also displayed in the [Find Results](#) panel.

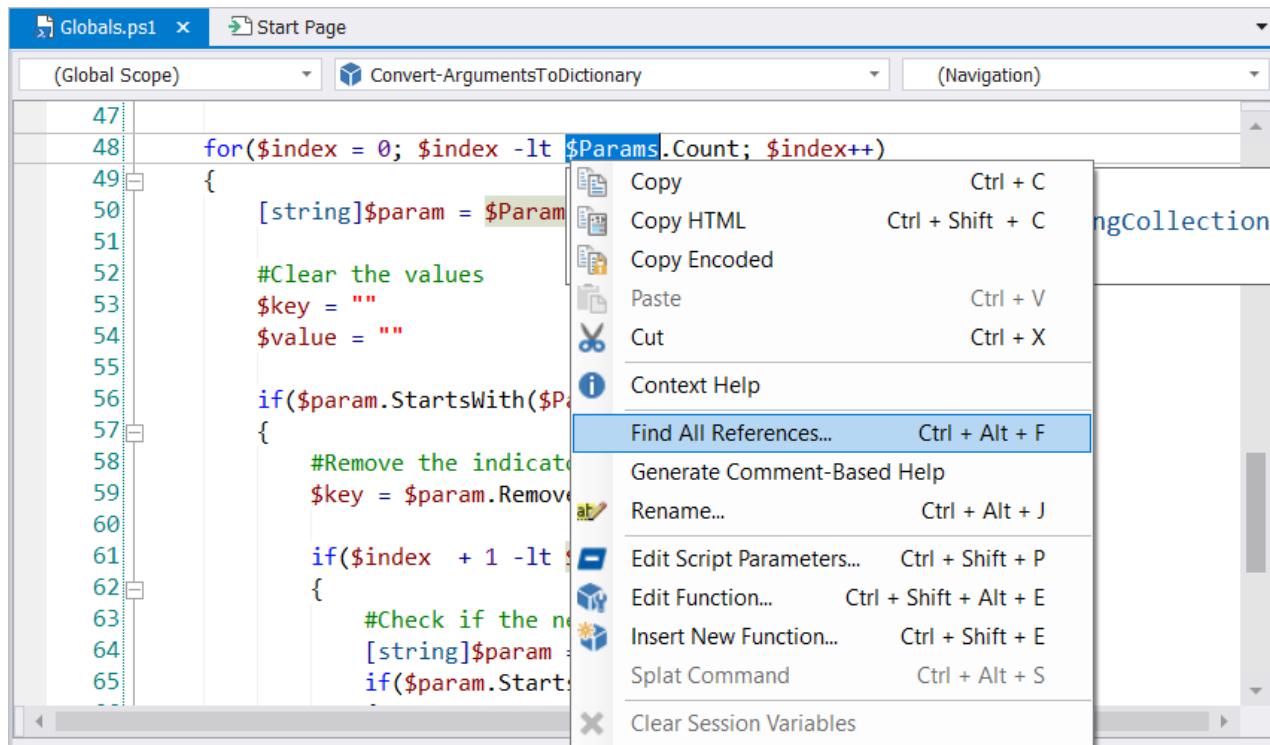
#### How to use Find All References

Click on the object, command, property, or method that you want to search for, and then on the ribbon select the **Home** tab > **Edit** section > **Find Replace** menu > **Find All References** (**Ctrl+Alt+F**):

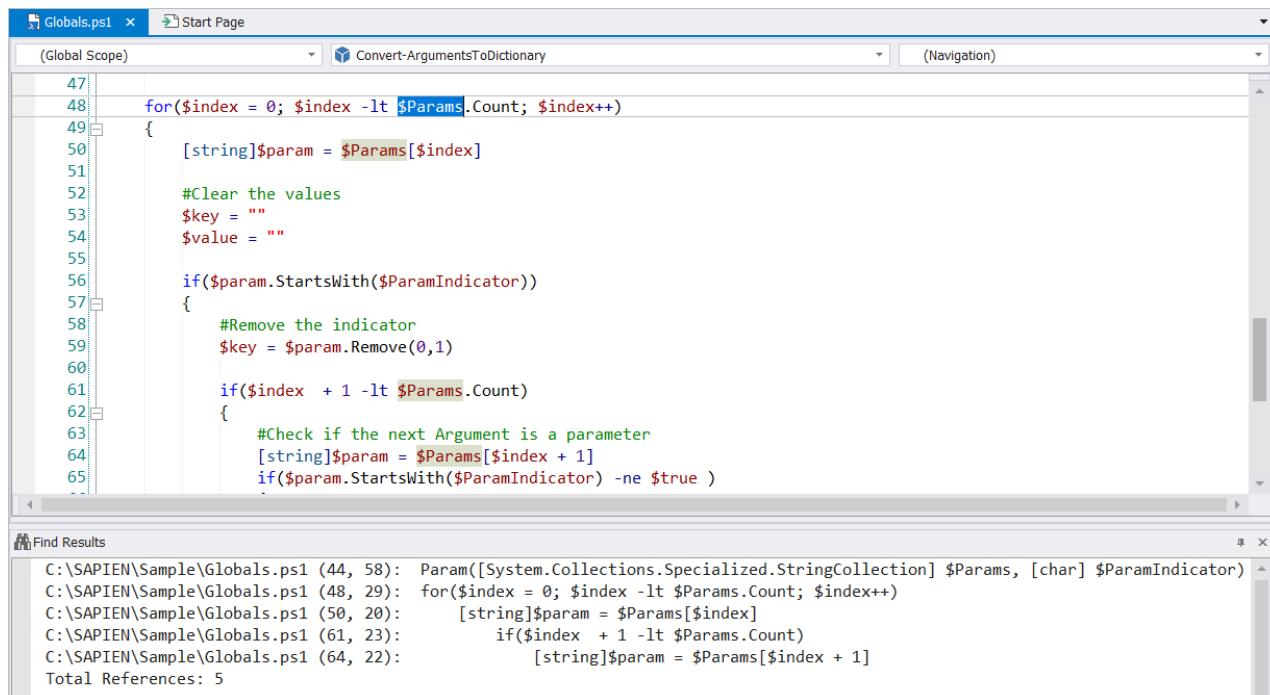


-OR-

Right-click on the object / command in the editor and select **Find All References** (**Ctrl+Alt+F**):



The references are highlighted in the editor and are also listed in the Find Results panel:



As previously mentioned ([Find in Files > Search Results](#)<sup>65</sup>), the Find Results panel allows you to view specific search results and track visited locations.

- i** If you are working with a project, references located in the other project files will also display in the Find Results panel.

## 5.5 PrimalSense™

PrimalSense™ is SAPIEN's brand-name for our powerful, flexible, contextually aware code-hinting and code-completion feature. PrimalSense is similar in functionality to Microsoft IntelliSense, which is found in Microsoft's Visual Studio products.

### PrimalSense Features

PrimalSense works automatically in most cases and provides the following features:

- **Syntax Coloring**

PrimalSense automatically colors your code syntax to help make literals, statements, comments, and other elements stand out more clearly. By default, PrimalSense doesn't begin working until you've typed a few characters from a recognized keyword, variable name, or other element.

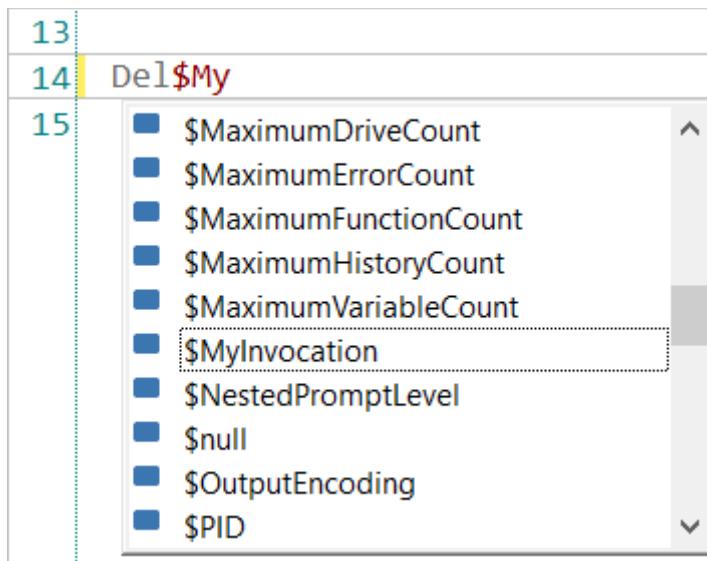
**Tip** To trigger immediate PrimalSense help, press **Ctrl+Space**, or type the first couple of letters of what you are looking for and then press **Ctrl+Space**. PrimalSense will display a list of intrinsic keywords, defined functions, cmdlets, variables, etc.

- **Case Correction**

When appropriate, PrimalSense automatically corrects the case of intrinsic statements, cmdlet names, variable names, and other elements.

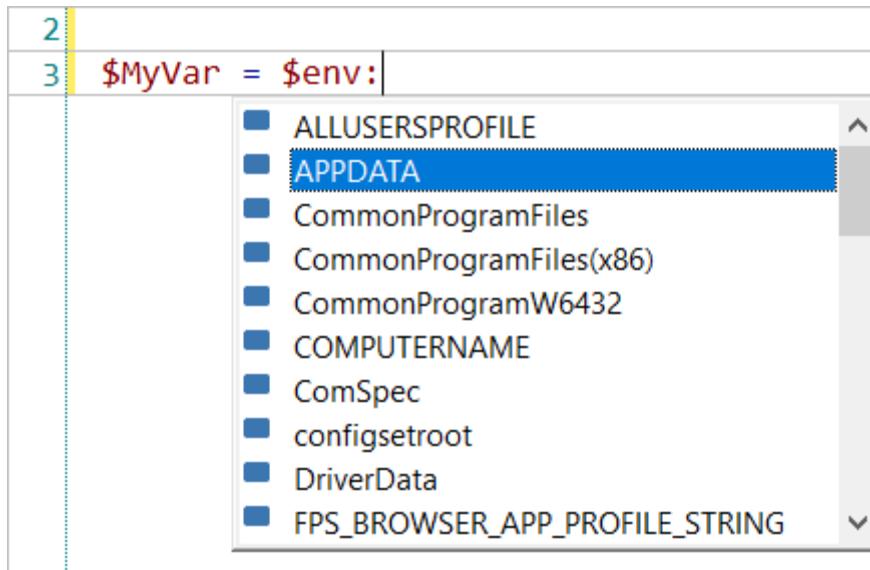
- **Variables**

PrimalSense automatically attempts to complete variable names, function names, and the names of other elements to save you from having to type the full names:

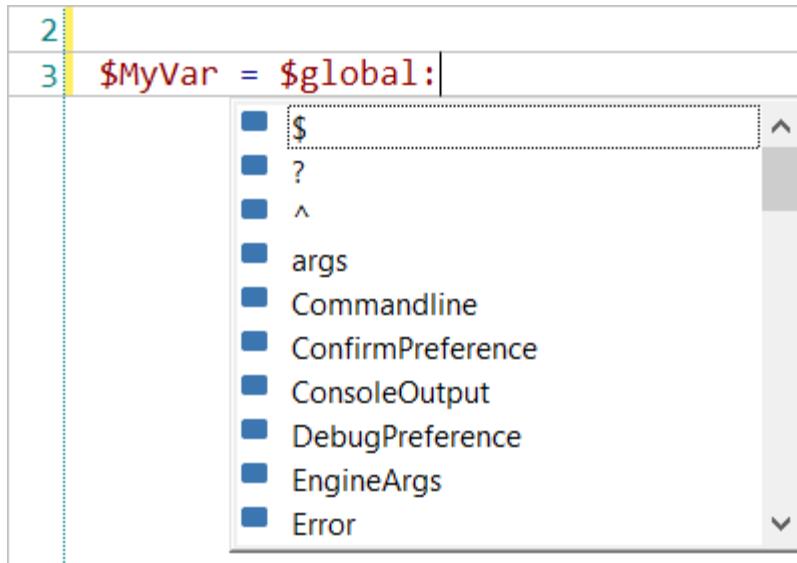


This also includes a number of standard PowerShell variables including:

- o \$env - to access the environment variables on your computer:



- o The scoping operators \$Global, \$Private and \$Script:



- **Code Completion**

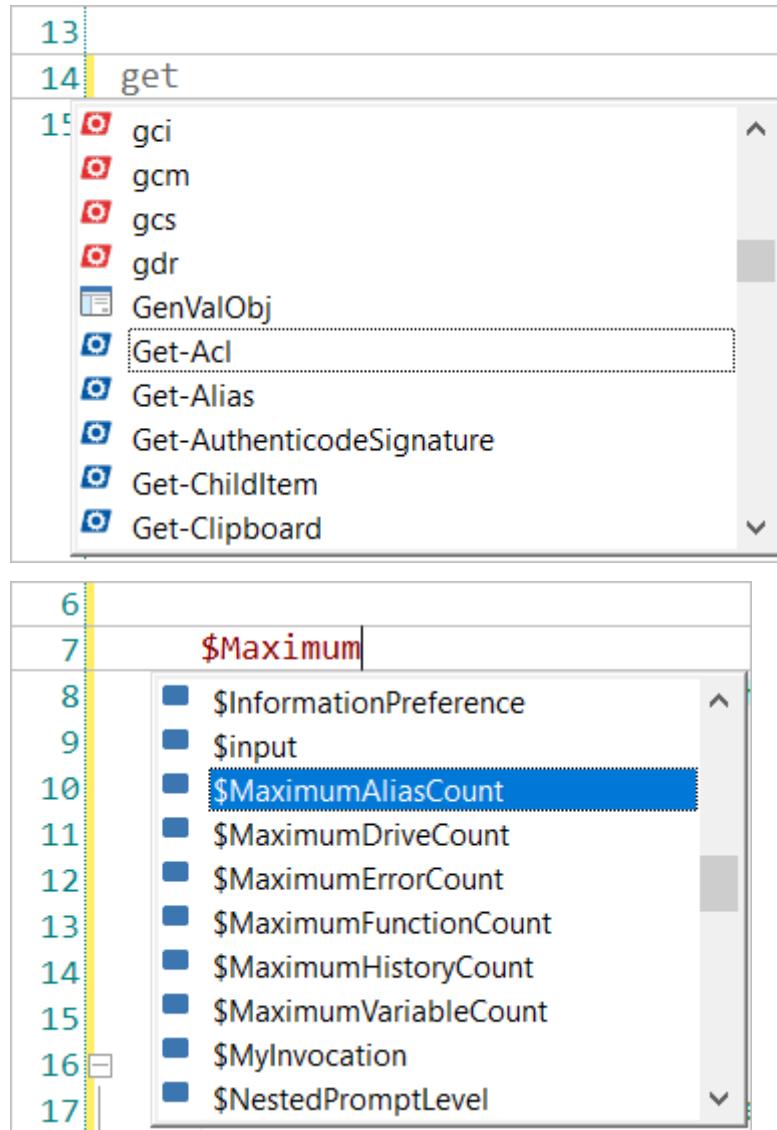
Code completion speeds the development process by predicting the rest of the construct as you are typing. The code editor will offer you choices as you type, thus reducing the amount you need to type and also minimizing the number of syntax errors in your scripts.

- **Cmdlet Help**

If you hover over any cmdlet or alias in the code editor PowerShell Studio will display a tool tip containing basic information about the cmdlet. Typing a space after the cmdlet or alias name will load the appropriate help file into the help panel.

- **Contextually Aware**

PrimalSense provides assistance while you are scripting, and also makes it easy to pick up right where you left off. Simply position the caret at the desired location and press **Ctrl+Space** to trigger PrimalSense. PrimalSense will list the appropriate items depending on the context:



PrimalSense for parameter attributes:

The screenshot shows a code editor window with the following PowerShell script:

```
13
14 Param (#$XPoints, $YPoints, $XTitle, $Title, $ChartSytle)
15     [ValidateNotNull()]_
16     [Parameter(Position=1,Mandatory=$true,)]
```

A tooltip is displayed on the right side of the screen, listing various parameters for the `Parameter` attribute:

- DontShow
- HelpMessage
- HelpMessageBaseName
- HelpMessageResourceId
- ParameterSetName
- ValueFromPipeline
- ValueFromPipelineByPropertyName
- ValueFromRemainingArguments

## Property and Method Completion

### Type Sense

When working with objects, PrimalSense displays pop-up lists containing available members, such as methods and properties. In many cases PrimalSense can provide "deep" assistance, helping you work with sub-objects and their members as well.

The screenshot shows a code editor window with the following PowerShell script:

```
15 $service = Get-Service
16 $Service[0].Di
```

A tooltip is displayed on the right side of the screen, listing available properties for the `Di` method:

- Continue
- CreateObjRef
- DependentServices
- DisplayName
- Dispose
- Equals
- ExecuteCommand
- GetHashCode
- GetLifetimeService
- GetType

The `DisplayName` property is selected in the list. A detailed tooltip for `DisplayName` is shown on the right:

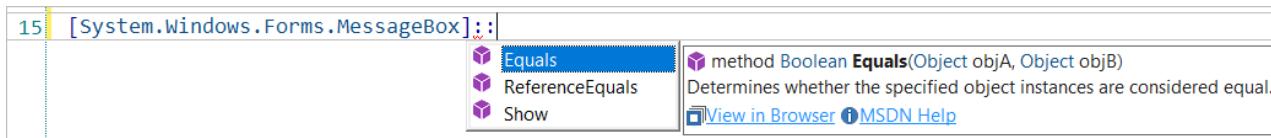
**property DisplayName**  
**System.String**  
Gets or sets a friendly name for the service.  
[View in Browser](#) [View Type](#) [MSDN Help](#)

PrimalSense also displays a tool tip showing more information about a property or method such as:

- Description.
- Parameter details.
- Link to view in the object browser.
- Link to view the online MSDN documentation.

## Method Completion

As you type the name of a method, PrimalSense will show you a list of all of the methods that start with the letters you have typed. If it has identified the correct method, press < Tab > to automatically enter the rest of the name:

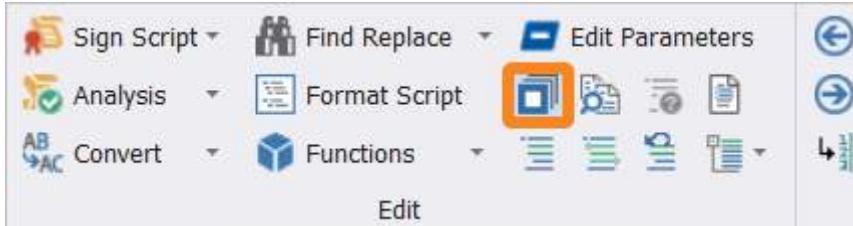


## Assemblies

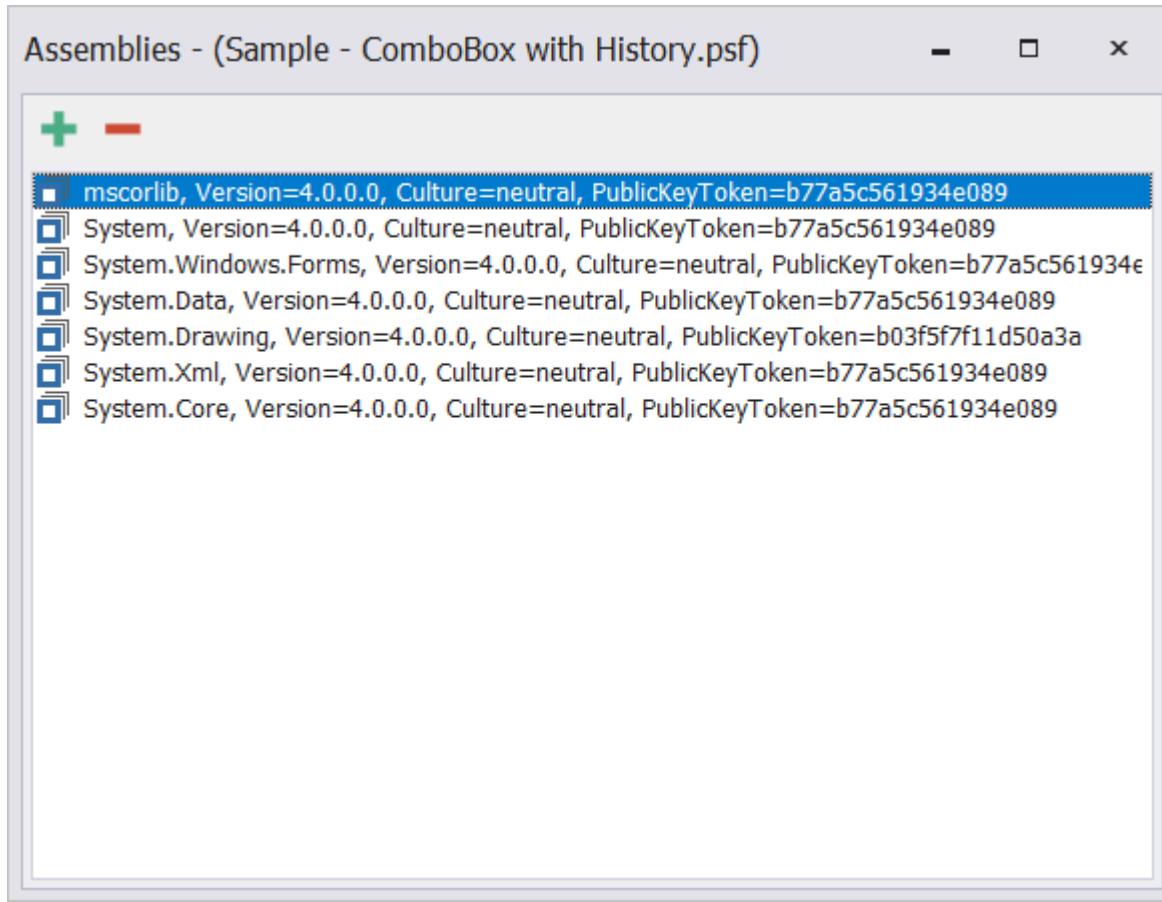
When you work with .NET assemblies in your scripts PowerShell Studio can parse them in order to offer better PrimalSense help. This also ensures that PowerShell uses the underlining types, such as the form controls.

Use either of these options to load assemblies into PowerShell Studio:

- On the Home tab > in the Edit section, click the Assemblies button:



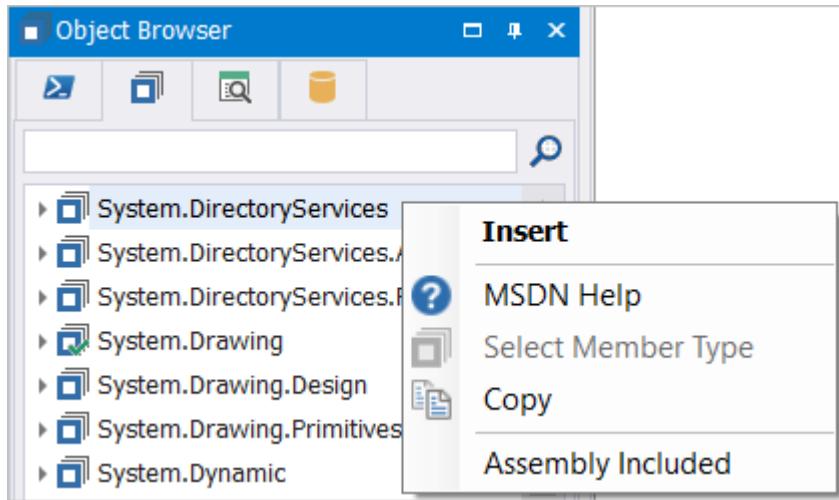
In the Assemblies dialog, use the green + button to load an assembly into the editor, or the red - button to remove an assembly:



- i** PowerShell Studio does not list the base .NET assemblies such as msclib, system, system.data, etc. As a result, the assembly list will most likely be empty but PowerShell Studio will continue to provide PrimalSense for these assemblies without explicitly listing them. This way, only the external assemblies that you specify will be listed. Note that GUI psf files created with PowerShell Studio 2017 or earlier will still retain the old assemblies list.

-OR-

- Right-click on a class in the Object Browser:



Choose one of the following options:

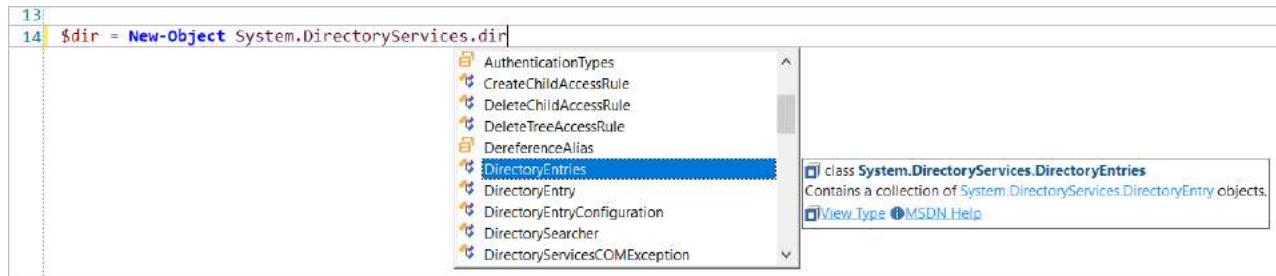
- **Insert**

Loads the containing assembly and adds the name of the class into the code editor.

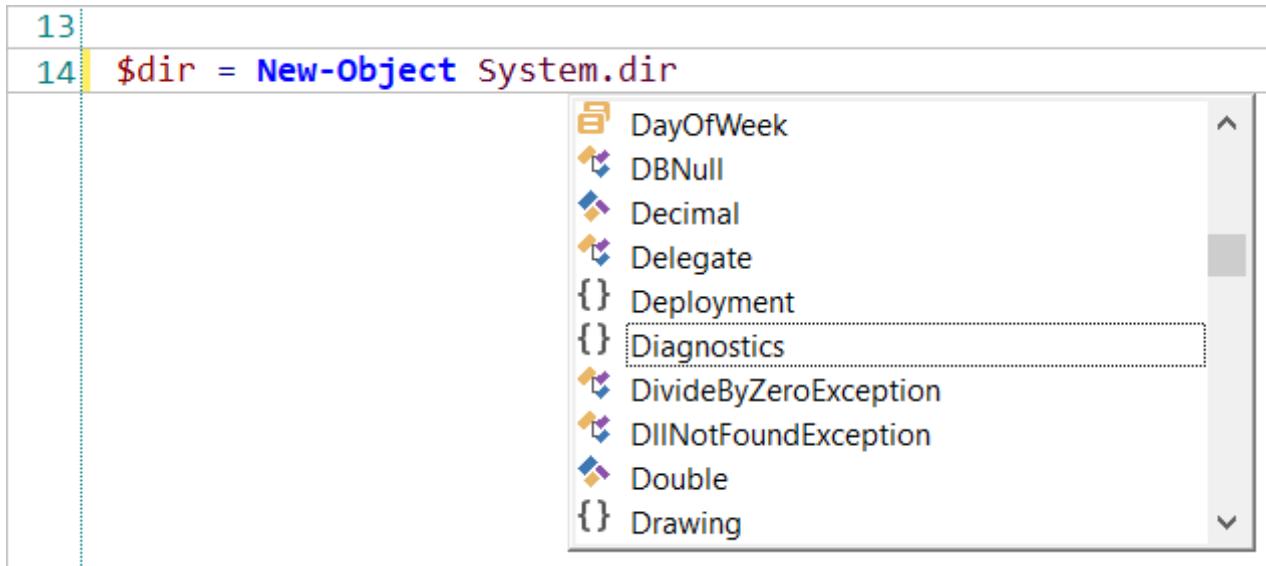
- **Assembly Included**

Loads or unloads the assembly.

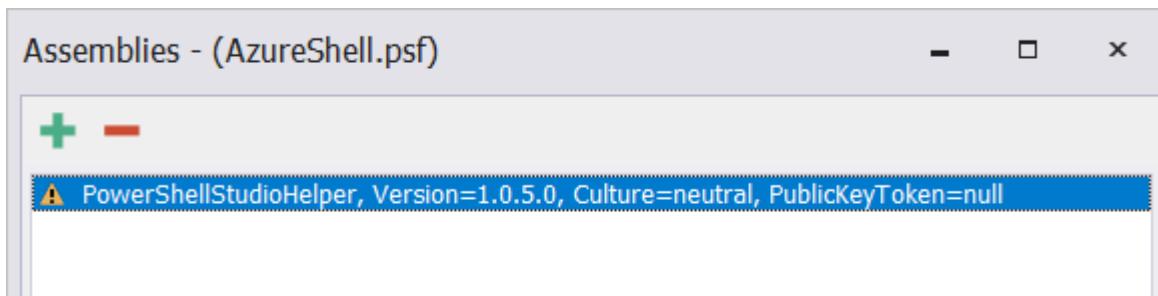
In this example the 'System.DirectoryServices' assembly has been added to PowerShell Studio and PrimalSense is able to offer help about the 'DirectoryEntry' class:



If the assembly is not added to PowerShell Studio, PrimalSense cannot provide any help. In the screenshot below, PrimalSense cannot offer any help because the 'System.DirectoryServices' assembly has not been loaded.



- i If an assembly has been removed from a machine, PowerShell Studio will not be able to load it the next time it starts. The Assemblies dialog indicates this by displaying a yellow triangle with an exclamation (!) point to the left of the assemblies that could not be loaded:



## 5.6 Converting Cmdlets and Aliases

PowerShell Studio provides options for converting PowerShell aliases into full cmdlet names, and for converting full cmdlet names into aliases.

### To convert a PowerShell alias into the full cmdlet name

Type a known PowerShell alias in the editor and then press <Tab>. The alias will be expanded into the full cmdlet name.

For example, if you type the alias *ps* into the editor:



Pressing the < Tab > key after the *ps* alias expands it into its full cmdlet name, *Get-Process*:

```
3
4 Get-Process
```

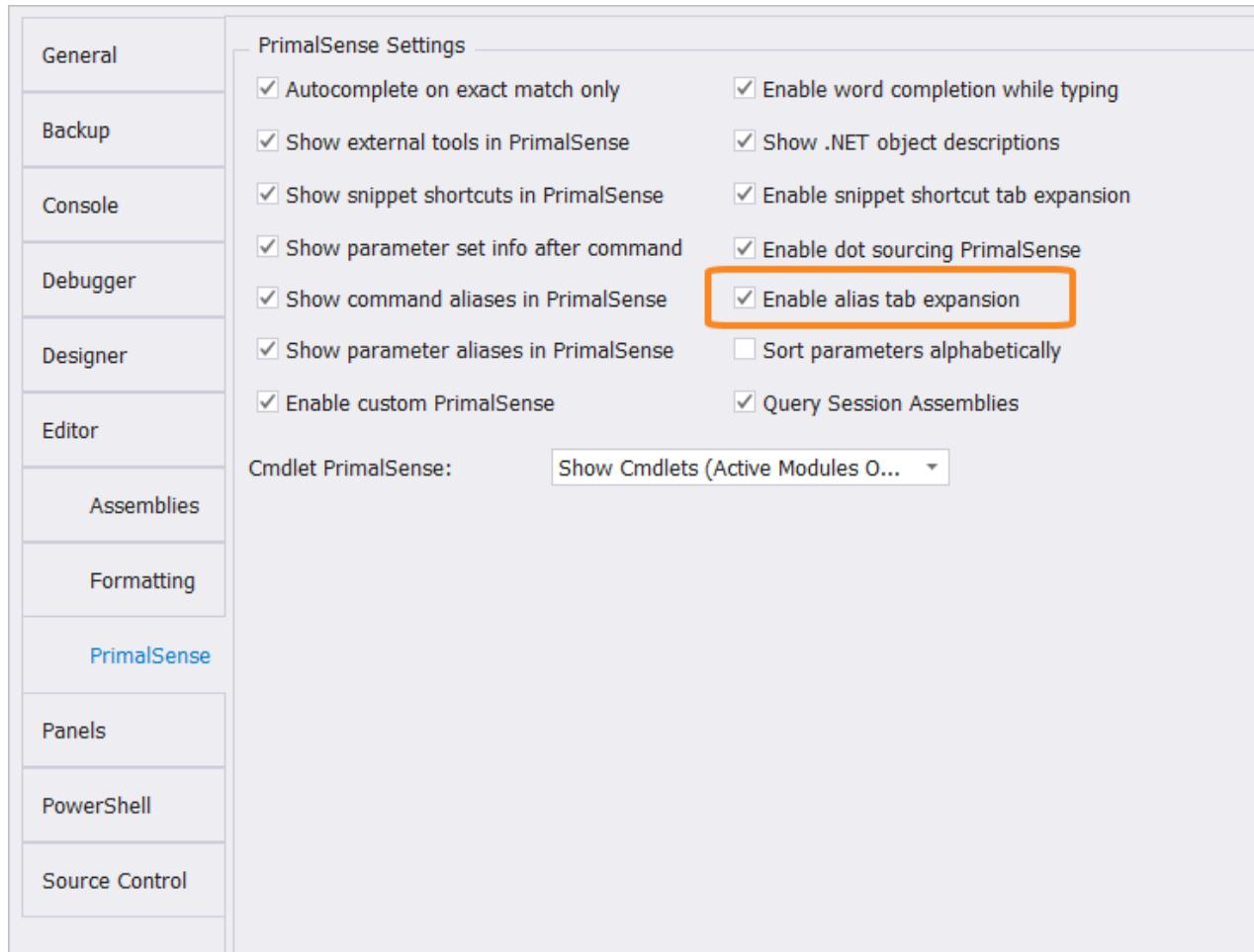
In a similar fashion, PrimalSense will expand parameter aliases into full names. If you type the alias -*cn* into the editor:

```
3
4 Get-Process -cn
```

Pressing the < Tab > key after the parameter alias *Cn* expands it into its full parameter name **ComputerName**:

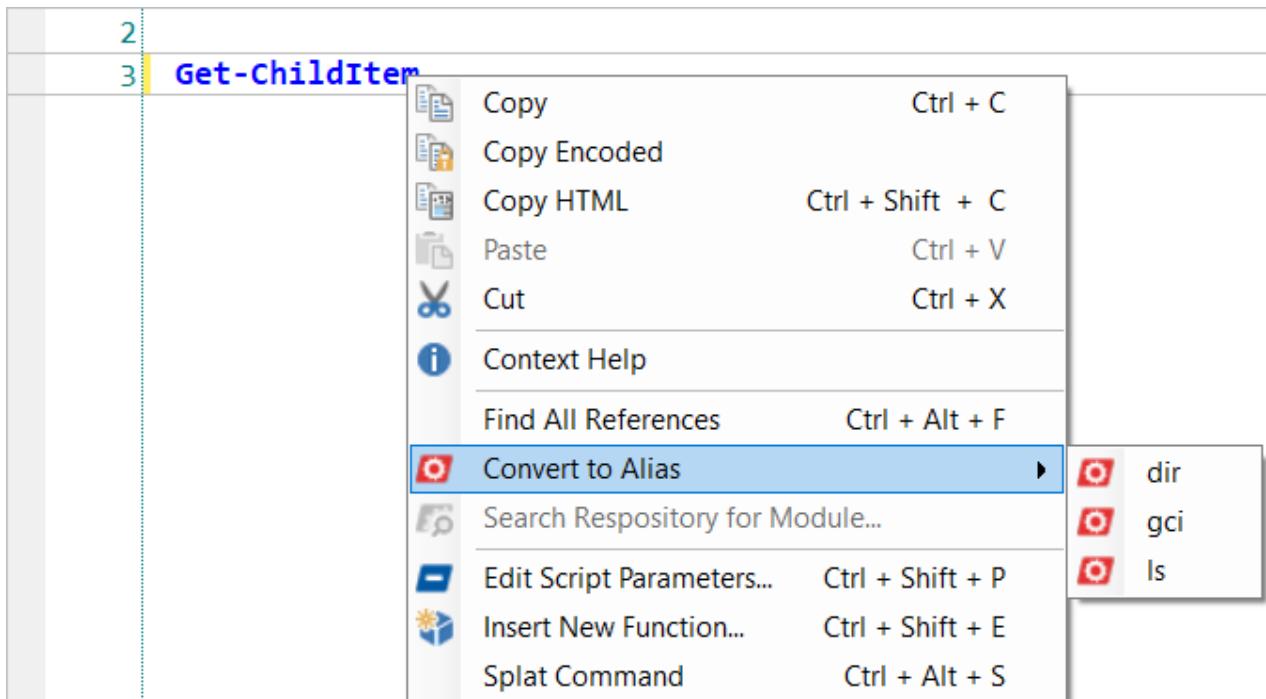
```
3
4 Get-Process -ComputerName
5
```

You can enable or disable alias expansion in Home > Options > Editor / PrimalSense > **Enable alias tab expansion**:



### To convert a full cmdlet name into alias

Right-click on the cmdlet name in the code editor and choose **Convert to Alias**. If a cmdlet has more than one alias, PowerShell Studio will list them:



PowerShell Studio provides two keyboard shortcuts for working with aliases:

- **Ctrl+Shift+A**

Convert all of the aliases in the current code file to their full cmdlet names.

- **Ctrl+B**

Toggle the alias on the current line between full name and alias.

 Aliases are great for minimizing the amount you need to type in the shell. When you are writing scripts however, it is best to expand all of the aliases. This makes it easier for less experienced colleagues to understand your scripts, and greatly facilitates debugging.

## 5.7 Snippets

PowerShell Studio provides a collection of snippets to help you complete common coding tasks quickly. You can also use the [Snippet Editor](#) 326 to create and edit snippets.

### About Snippets

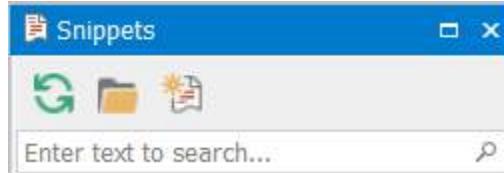
Snippets are small pieces of reusable code that can be quickly inserted into your scripts, thus saving you time and reducing errors. This piece, or "snippet" of code, can vary from a full-fledged function to a simple single line statement. Snippets come in a variety of languages such VBScript, PowerShell, C#, etc.

PrimalScript and PowerShell Studio come with extensive libraries of reusable code snippets. You can also save any text or code block as a snippet to automate code development. Snippets can include

placeholders; PrimalScript and PowerShell Studio will prompt you to supply values for these when you use the snippet.

## Snippets Panel

Use the Snippets panel to access and manage snippets:



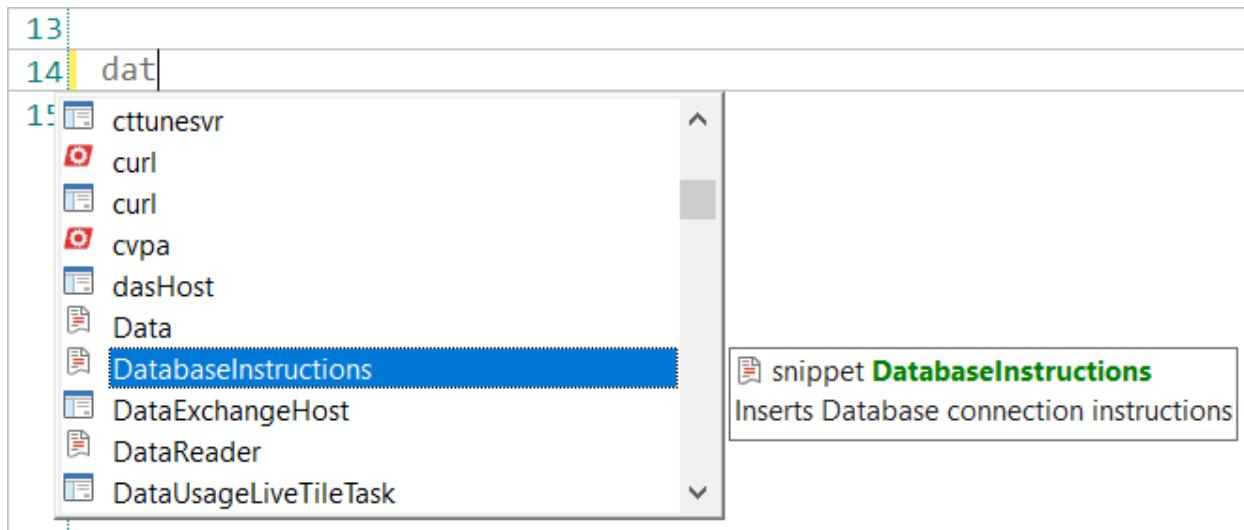
### To access the Snippets panel:

- On the Home ribbon, in the Windows section, select Snippets from the Panels drop-down menu.
- OR-
- Chorded keyboard shortcut: Press **Ctrl+Alt+P**, release, then press **S**

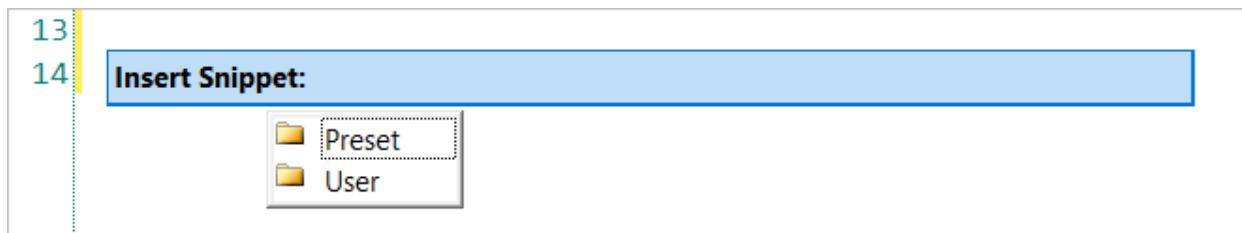
## Inserting Snippets

PowerShell Studio provides many options for inserting snippets into your code. The snippet will be inserted at the current caret position:

- Type the first few letters of the snippet name in the code editor and PrimalSense will display a list of options. Arrow up or down to select the desired snippet, and then press <Tab> to insert it into your file:



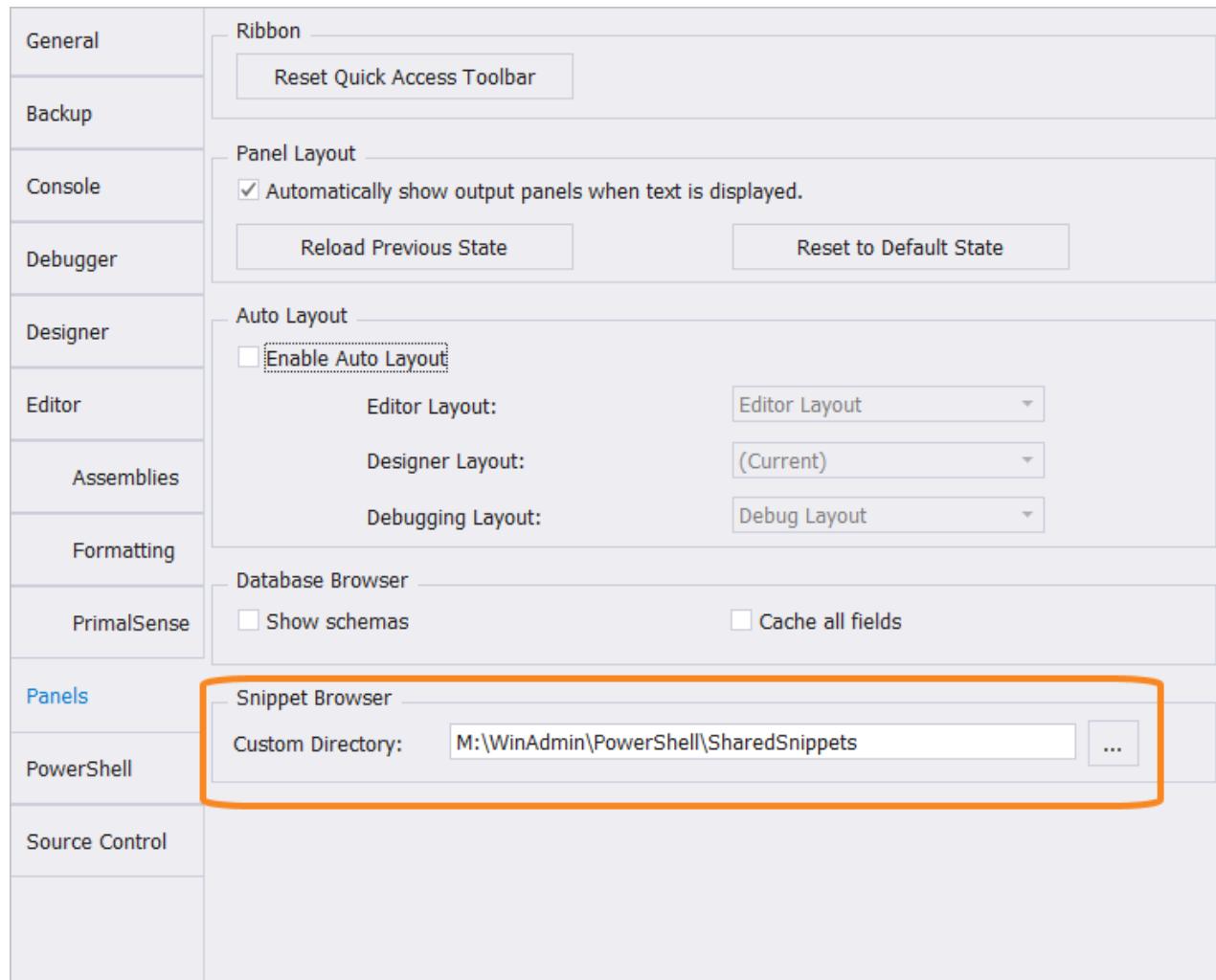
- Right-click (**Ctrl+K**) in the code editor and PowerShell Studio will display a snippet selector. Navigate through the snippet folders using the **Up / Down Arrow** keys and the **<Enter>** key to select a snippet:



- Follow any of these options to add a snippet to a script from the [Snippets panel](#):
  - Right-click on a snippet and select **Insert**.
  - Double-click a snippet.
  - Click the snippet and then press **<Enter>**.
  - Drag a snippet and drop it in the code editor.

## Custom Snippet Folder

You can add a custom directory to the Snippets panel, such as a network share or a local directory, via **Home > Options > Panels**. Specify the folder path in the Custom Directory field:



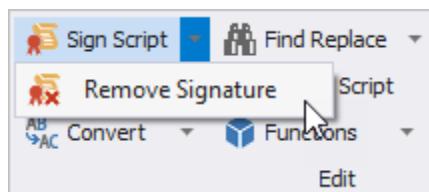
- i** New snippets that you create will automatically be stored in the 'User' folder and are stored in:  
%Users%\<user>\AppData\Roaming\SAPIEN\User Snippets\PowerShell

## 5.8 Script Signing

Script signing is the process of adding a digital signature to scripts.

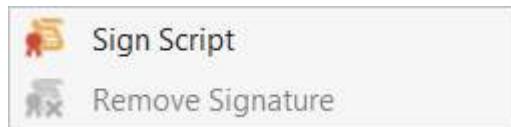
### How to Sign a Script or Remove a Signature

You can sign a script or remove a signature from the Home > Edit > Sign Script menu:



-OR-

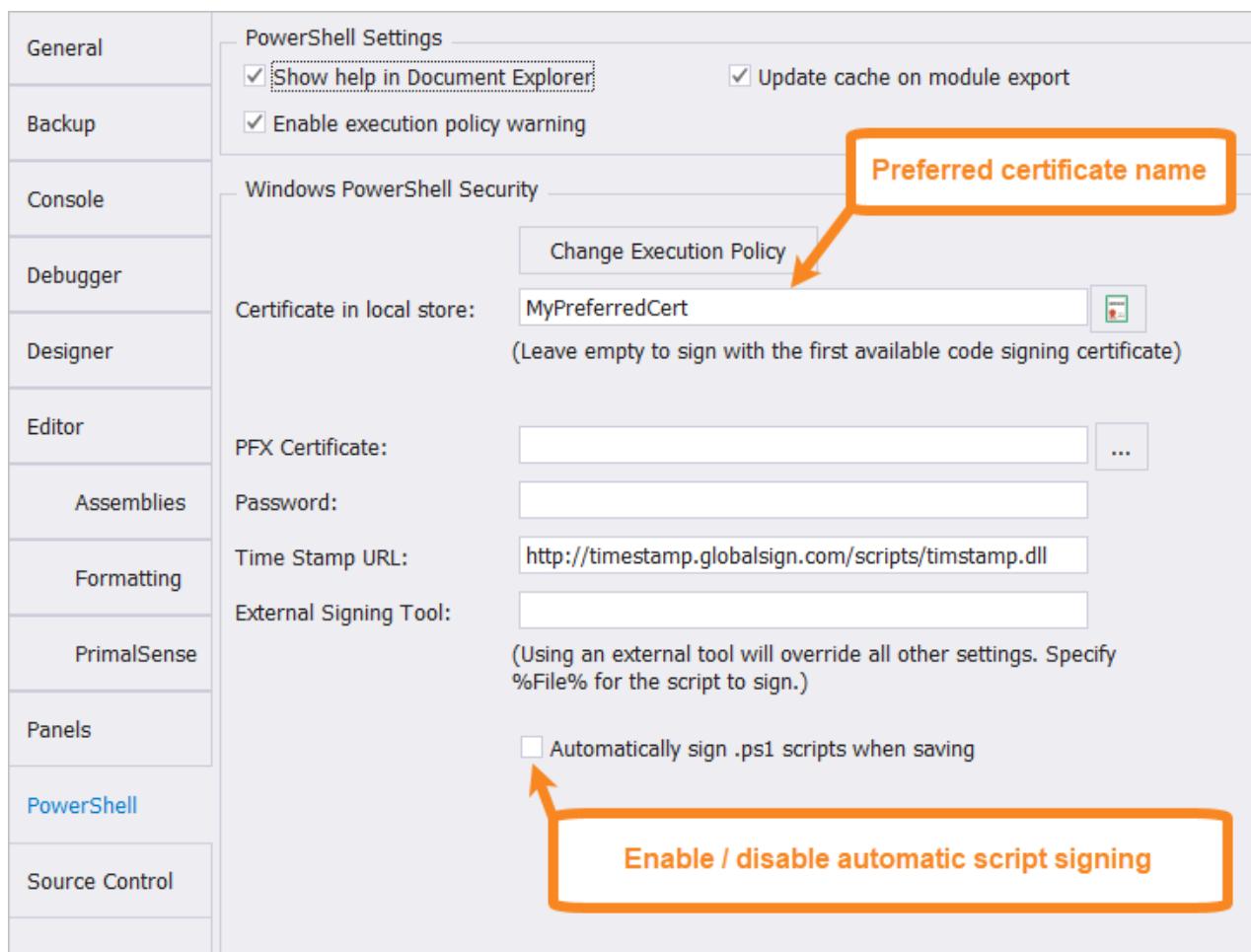
Right-click on the tab containing the file name at the top of the script editor:



- **Sign Script**  
Sign a script.
- **Remove Signature**  
Remove an existing signature.

## Script Signing Options

PowerShell Studio can automate script signing so that scripts are signed when saved or exported. Script signing options can be configured in **Home > Options > PowerShell > Windows PowerShell Security**:



## To enable automatic script signing

- Select the **Automatically sign .ps1 scripts when saving** checkbox at the bottom of the dialog.

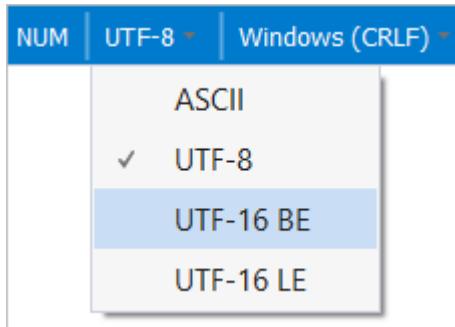
## To designate a preferred code signing certificate

- Enter the certificate name in the **Certificate in local store** field.

 PowerShell Studio also supports PFX format certificates.

## 5.9 File Encoding

File encoding can be changed by selecting an option from the file encoding menu on the status bar:



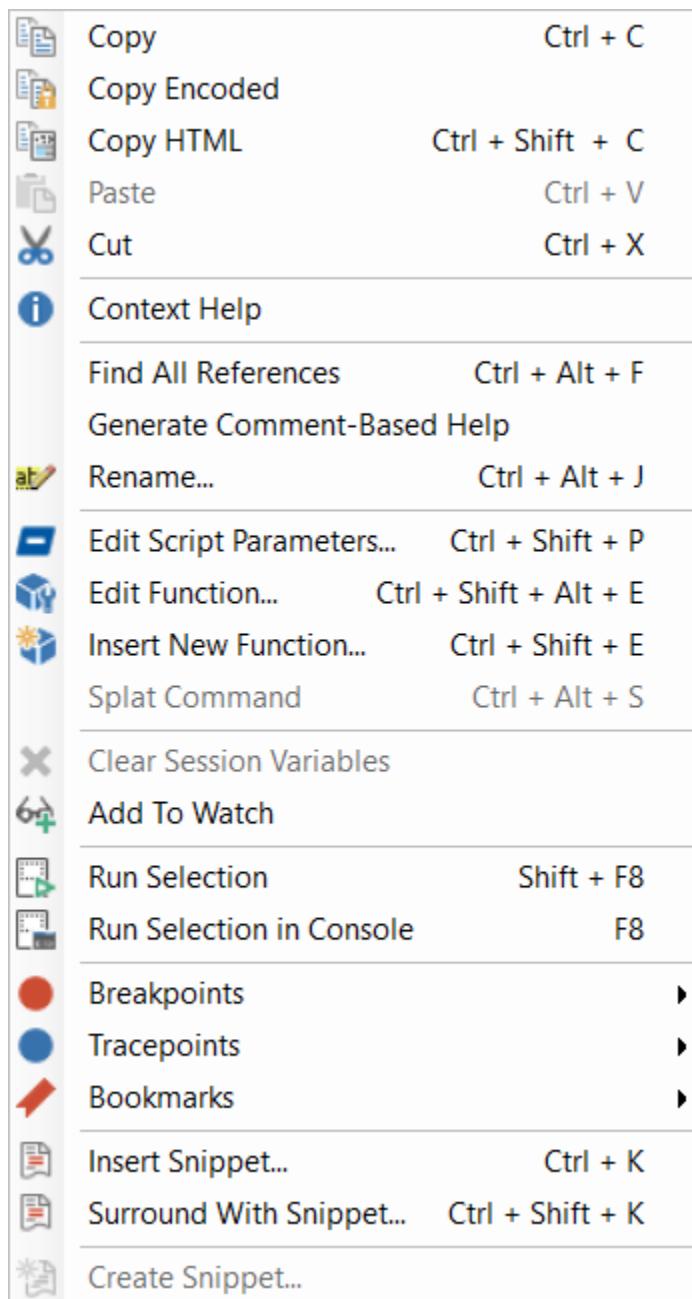
- [ASCII](#)
- [UTF-8](#)
- [UTF-16 BE](#)
- [UTF-16 LE](#)

 UTF-8 is the default and supports international characters.

## 5.10 Context Menu Options

Many of the functions available on the ribbon bar can also be accessed by right-clicking in the code editor.

## Script Editor - Context Menu Options



- The context menu options will vary depending on the file type, the location of the cursor in the document, and also if any characters are highlighted. For example, right-clicking within a document that is part of a Project will reveal project-related options, while right-clicking on a 'cmdlet' enables the 'Convert to Alias' menu option.

## 5.11 Functions and Parameters

PowerShell Studio provides tools that make it easy to quickly create functions, parameters, and parameter sets—the [Function Builder](#)<sup>[85]</sup> and the [Parameter Builder](#)<sup>[108]</sup>.

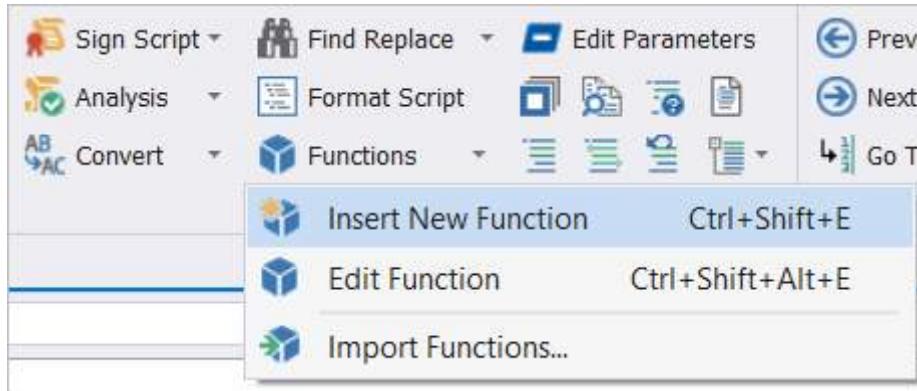
### 5.11.1 Function Builder

The Function Builder is a robust tool that simplifies the creation of complex functions. It is a time saver and can also be a learning tool for those who may not be familiar with the intricacies of creating an advanced function. This section covers creating and editing functions, including parameters and parameter sets.

#### To launch the Function Builder

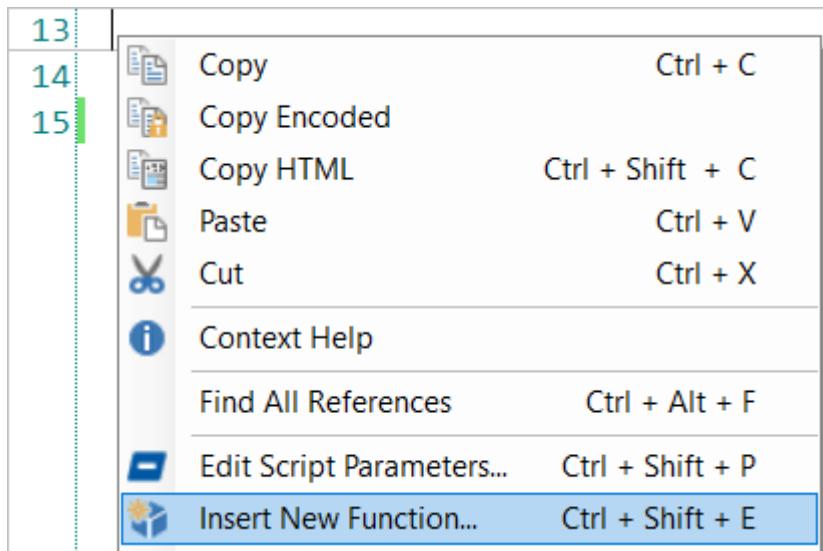
Position the caret in the code editor where you want to insert a new function, and then use one of these options:

- On the **Home** tab > in the **Edit** section, click the **Functions** drop-down > then select **Insert New Function (Ctrl+Shift+E)**:



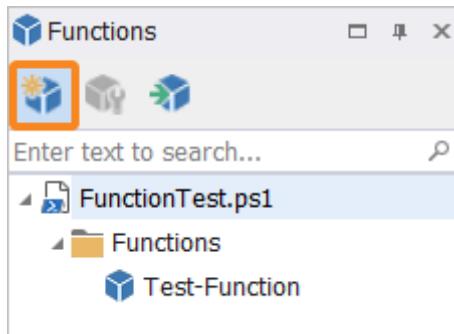
-OR-

- In the code editor, right-click and select **Insert New Function (Ctrl+Shift+E)**:

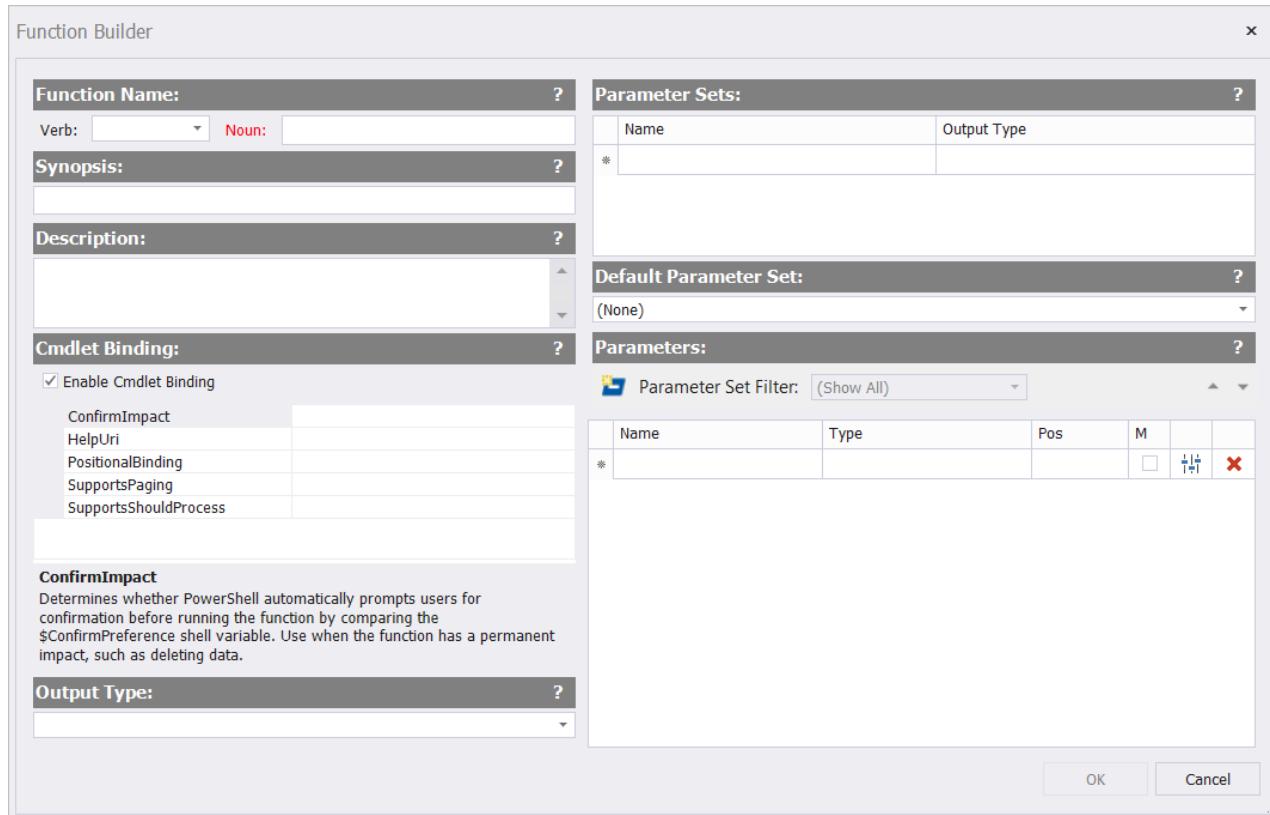


-OR-

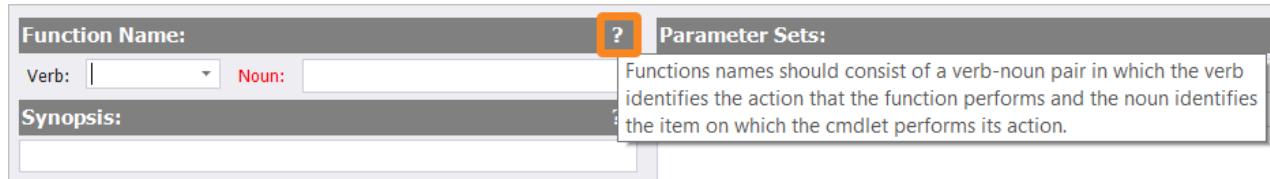
- In the Functions panel click the Insert New Function button:



The Function Builder dialog will launch:



If you hover over or click over a question mark, a pop-up help message will explain the respective field:



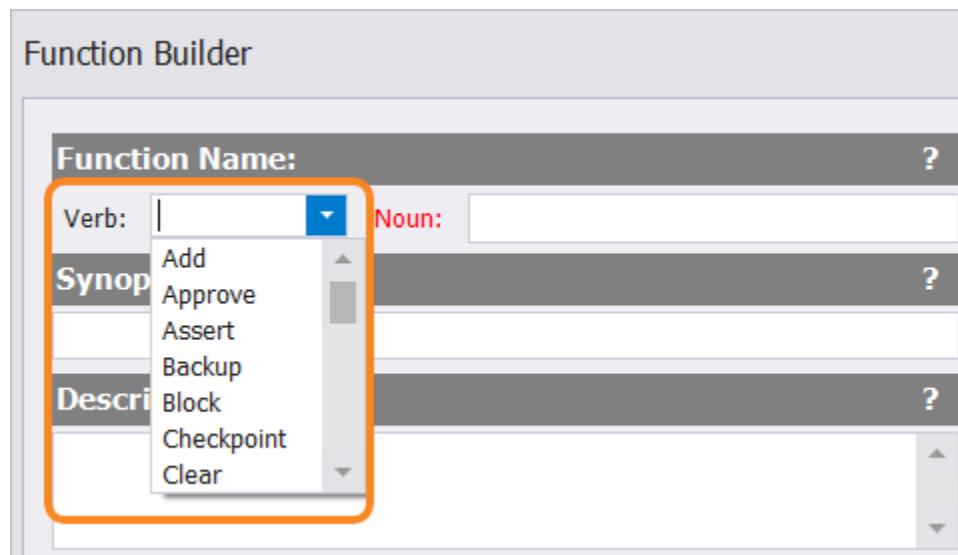
We will explore each field of the Function Builder in the following topics:

- [Function \(Cmdlet\) Name](#) <sup>88</sup>
- [Synopsis and Description](#) <sup>88</sup>
- [Cmdlet Binding](#) <sup>88</sup>
- [Output Type](#) <sup>89</sup>
- [Parameter Sets](#) <sup>90</sup>
- [Default Parameter Set](#) <sup>91</sup>
- [Parameters](#) <sup>91</sup>
- [Parameter Set Filter](#) <sup>95</sup>
- [Parameter Editor](#) <sup>97</sup>
- [Special Considerations](#) <sup>102</sup>

## 5.11.1.1 Function (Cmdlet) Name

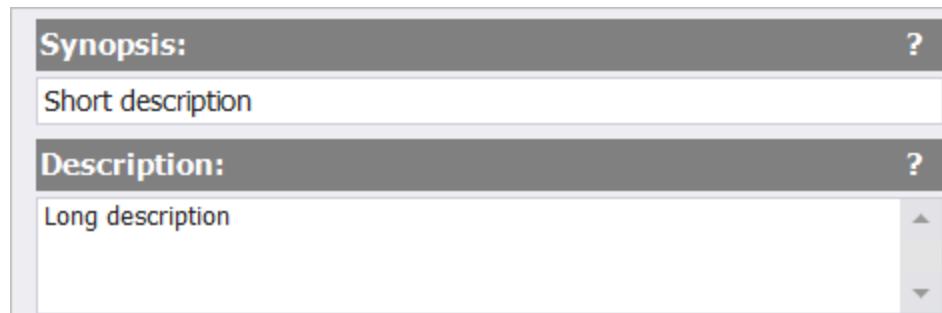
The Function Builder provides fields for a Verb and a Noun.

The Verb field contains a combo-box with a list of approved verbs, and also provides the option to enter an unapproved verb if necessary:



## 5.11.1.2 Synopsis and Description

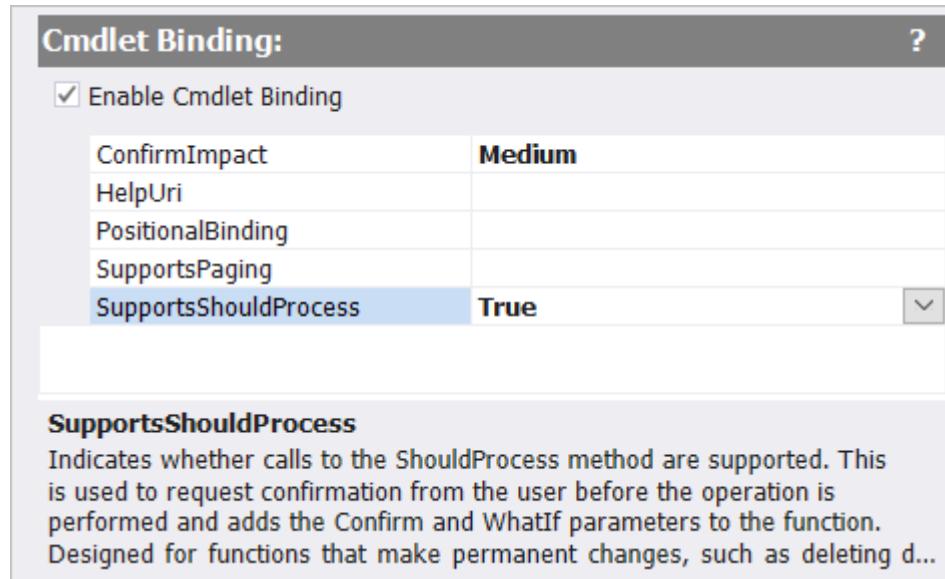
Enter a quick synopsis and a longer description in the Synopsis and Description fields:



- i The information provided here is used to generate the function's Comment-based Help.

## 5.11.1.3 Cmdlet Binding

Check the **Enable Cmdlet Binding** option if you want your function to behave like a cmdlet and take advantage of PowerShell's built-in parameters such as *Debug* and *Verbose*, and/or process input from the pipeline. Set the cmdlet binding attributes by using the properties grid:



- 👉 The property grid displays a Help description when you click on an attribute.

The Cmdlet Binding option will add the [CmdletBinding] attribute to your function:

```
28 function Get-Data
29 {
30     [CmdletBinding(ConfirmImpact = 'Medium',
31                 SupportsShouldProcess = $true)]
32     param ()
33
34     if ($pscmdlet.ShouldProcess("Target", "Operation"))
35     {
36         #TODO: Place script here
37     }
38 }
```

#### 5.11.1.4 Output Type

The Output Type combo-box allows you to specify the output type of the function by selecting from existing types or entering a type:



The appropriate attribute will be added to your code:

```
28 function Get-Data
29 {
30     [CmdletBinding(ConfirmImpact = 'Medium',
31                   SupportsShouldProcess = $true)]
32     [OutputType([string])]
33     param ()
34
35     if ($pscmdlet.ShouldProcess("Target", "Operation"))
36     {
37         #TODO: Place script here
38     }
39 }
```

**i** PowerShell Studio also provides PrimalSense code completion for types and namespaces as you enter content in the Output Type field.

## 5.11.1.5 Parameter Sets

The Function Builder allows you to define parameter sets, assign parameter sets to individual parameters, designate a default parameter set, and much more.

### To define a parameter set

Type a name for the parameter set in the **Name** column. You can also specify an (optional) output type for each parameter set:

Parameter Sets:	
Name	Output Type
Name Set	string
*	

Defining a parameter set enables additional options within the Function Builder:

#### *Before a Parameter Set is defined*

Parameter Sets:						
Name	Output Type					
*						
Default Parameter Set:						
(None)						
Parameters:						
Parameter Set Filter: (Show All)						
Name	Type	Pos	M			
*				<input type="checkbox"/>		

#### *Options enabled after a Parameter Set is defined*

Parameter Sets:						
Name	Output Type					
Name Set						
*						
Default Parameter Set:						
Name Set						
Parameters:						
Parameter Set Filter: (Show All)						
Name	Type	Parameter Set	Pos	M		
*				<input type="checkbox"/>		

#### 5.11.1.6 Default Parameter Set

Use the Default Parameter Set drop-down list to set the CmdletBinding attribute:



#### 5.11.1.7 Parameters

The Parameters section of the Function Builder is used for adding or editing parameters, and you can also filter the parameters displayed by using the [Parameter Set Filter](#) [95].

##### To add a parameter

- Name

Enter the name of the parameter:

Parameters:							?
		Parameter Set Filter:	(Show All)				
Name	Type	Parameter Set	Pos	M			
Name				<input type="checkbox"/>			
*				<input type="checkbox"/>			

- **Type**

Specify the object type of the parameter:

Parameters:							?
		Parameter Set Filter:	(Show All)				
Name	Type	Parameter Set	Pos	M			
Name	string			<input type="checkbox"/>			
	string						

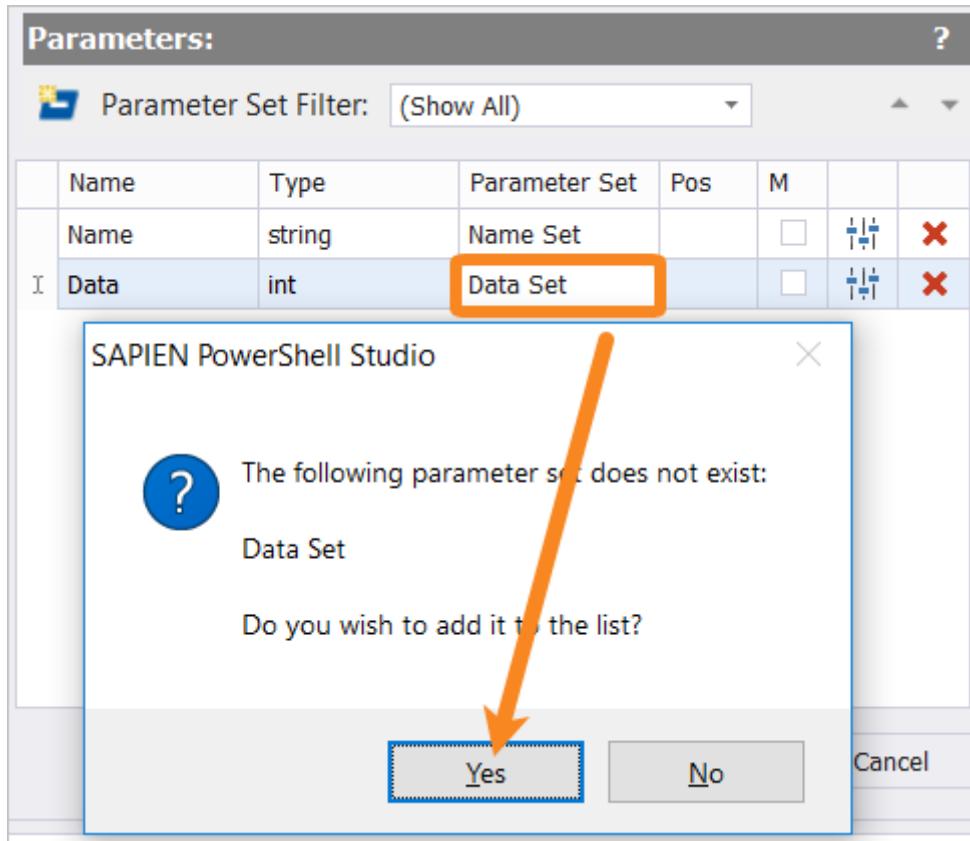
- **Parameter Set**

Assign the parameter to a parameter set:

Parameters:							?
		Parameter Set Filter:	(Show All)				
Name	Type	Parameter Set	Pos	M			
Name	string	Name Set		<input type="checkbox"/>			
		Name Set					

-OR-

Define a new parameter set:



Multiple parameter sets can be assigned to a single parameter by using a comma separator:

Name	Type	Parameter Set	Pos	M
Name	string	Name Set, Data Set	1,	<input checked="" type="checkbox"/>

- **Pos (Position)**

Define the parameter's position attribute (optional):

Name	Type	Parameter Set	Pos	M
Name	string	Name Set, Data Set	1, 2	<input checked="" type="checkbox"/>
Data	int	Data Set	1	<input checked="" type="checkbox"/>
FilePath	string[]	Name Set	2	<input type="checkbox"/>

By default, the parameter's position is determined by the ordering of the parameters.

For parameters that are assigned to multiple parameter sets, you can define each position by using a comma separator:

Name	Type	Parameter Set	Pos	M		
Name	string	Name Set, Data Set	1, 2	<input checked="" type="checkbox"/>		

- The position values should follow the same ordering as the parameter set assignments.

- **M (Mandatory)**

Define the parameter as mandatory (optional):

**Parameters:**

Parameter Set Filter: (Show All)

Name	Type	Parameter Set	Pos	M		
Name	string	Name Set, Data Set	1, 2	<input checked="" type="checkbox"/>		
Data	int	Data Set	1	<input checked="" type="checkbox"/>		
FilePath	string[]	Name Set	2	<input type="checkbox"/>		

### To edit a parameter

To edit advanced parameter settings, click the **Details** button (*Ctrl+E*):

**Parameters:**

Parameter Set Filter: (Show All)

Name	Type	Parameter Set	Pos	M		
Name	string	Name Set, Data Set	1, 2	<input checked="" type="checkbox"/>		

- For more information about editing parameter details, see [Parameter Editor](#).

### To remove a parameter

To remove a parameter, click the red X button:

Parameters:						
		Parameter Set Filter:		(Show All)		
	Name	Type	Parameter Set	Pos	M	
▶	Name	string	Name Set, Data Set	1, 2	<input checked="" type="checkbox"/>	 

#### 5.11.1.8 Parameter Set Filter

Use the Parameter Set Filter to view parameters by their assigned Parameter Set:

Parameters:						
		Parameter Set Filter:		(Show All)		
	Name	Type	Parameter Set	Pos	M	
▶	Name	string	(None)	Set 1, 2	<input checked="" type="checkbox"/>	 
	Data	int	Name Set	1	<input checked="" type="checkbox"/>	 
	FilePath	string[]	Data Set	2	<input type="checkbox"/>	 
	Info	string			<input type="checkbox"/>	 
	Active	switch			<input type="checkbox"/>	 
	EmpID	int			<input type="checkbox"/>	 
▶					<input type="checkbox"/>	 

- **Show All (Default)**

Show all of the parameters.

- **None**

Show only the parameters that are not assigned to a parameter set.

- < **Parameter Set Name** >

Show only the parameters assigned to < **Parameter Set Name** >.

#### To work with the parameters assigned to a specific parameter set

Select the filter for a specific parameter set:

**Parameters:**

Parameter Set Filter: Name Set

Name	Type	Parameter Set	Pos	M		
Name	string	Name Set, Data Set	1	<input checked="" type="checkbox"/>		
FilePath	string[]	Name Set	2	<input type="checkbox"/>		

**i** The position (Pos) field now only shows the parameter's position within the selected parameter set and no longer lists the other parameter set positions.

### To move or swap parameter positions

When the view is filtered for a particular parameter set, use the Up and Down menu buttons to move or swap the parameter positions:

**Parameters:**

Parameter Set Filter: Name Set

Name	Type	Parameter Set	Pos	M		
Name	string	Name Set, Data Set	1	<input checked="" type="checkbox"/>		
FilePath	string[]	Name Set	2	<input type="checkbox"/>		

**Parameters:**

Parameter Set Filter: Name Set

Name	Type	Parameter Set	Pos	M		
FilePath	string[]	Name Set	1	<input type="checkbox"/>		
Name	string	Name Set, Data Set	2	<input checked="" type="checkbox"/>		

**i** When the default (Show All) filter is selected:

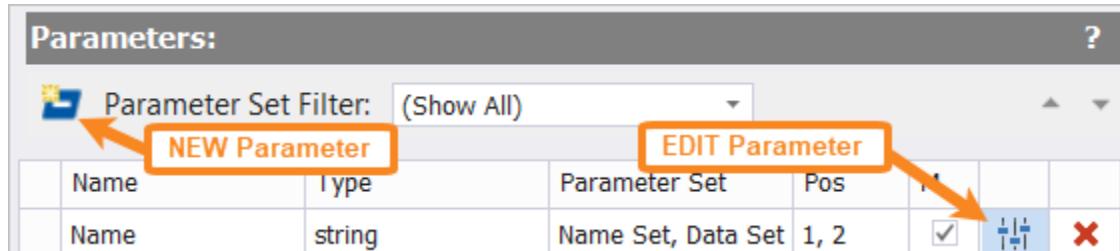
- The Up and Down move buttons only change the order in which the parameters are declared in the function.
- The Mandatory (M) check box refers to the parameter's first assigned parameter set.

When a parameter set is declared, the generated function will now include a switch statement to help you differentiate between the parameter sets:

```
switch ($PsCmdlet.ParameterSetName)
{
    'Name Set' {
        #TODO Place script here
        break
    }
    'Data Set' {
        #TODO Place script here
        break
    }
}
```

### 5.11.1.9 Parameter Editor

Use the Parameter editor dialog to create a new parameter or edit the details of an existing parameter, such as adding validation and aliases.

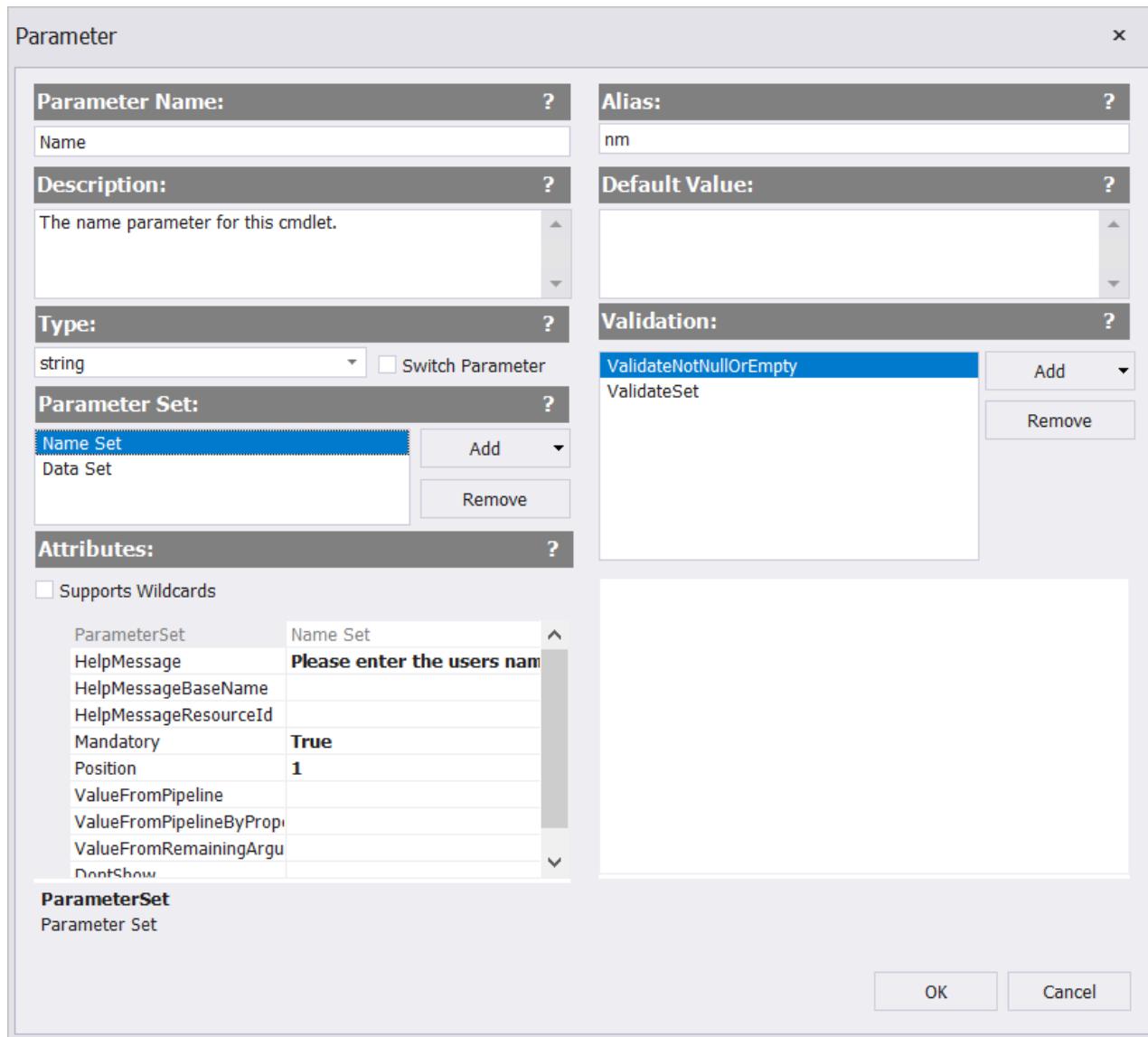


#### To create a new parameter

Click the **New Parameter** button (*Ctrl+N*).

#### To edit a parameter

Click the **Edit Parameter** button (*Ctrl+E*) on the row of the parameter that you want to edit.



Parameter dialog field descriptions:

- **Parameter Name**

The name of the parameter.

- **Description**

The description of the parameter that will appear in the Help comment.

- **Type**

Specify the object type of the parameter.

**i** Checking **Switch Parameter** will make the parameter a switch parameter (the same as typing 'switch' in the type field).

- **Parameter Set**

Designate the Parameter Set, or add a new Parameter Set:



- **Attributes**

Set the parameter's attributes, such as marking the parameter as Mandatory:

The 'Attributes' dialog for a parameter named 'NameSet'. The 'Mandatory' attribute is set to 'True'. A tooltip for 'Mandatory' explains it indicates whether the parameter is required when the cmdlet or function is run. Other attributes shown include HelpMessage, HelpMessageBaseName, HelpMessageResourceId, and Position.

ParameterSet	NameSet
HelpMessage	<b>Please enter the users name</b>
HelpMessageBaseName	
HelpMessageResourceId	
Mandatory	<b>True</b>
Position	
ValueFromPipeline	
ValueFromPipelineByPropertyName	
ValueFromRemainingArguments	
DontShow	

**Mandatory**  
Indicates whether the parameter is required when the cmdlet or function is run.

💡 A Help description is displayed when editing a parameter attribute.

- **Alias**

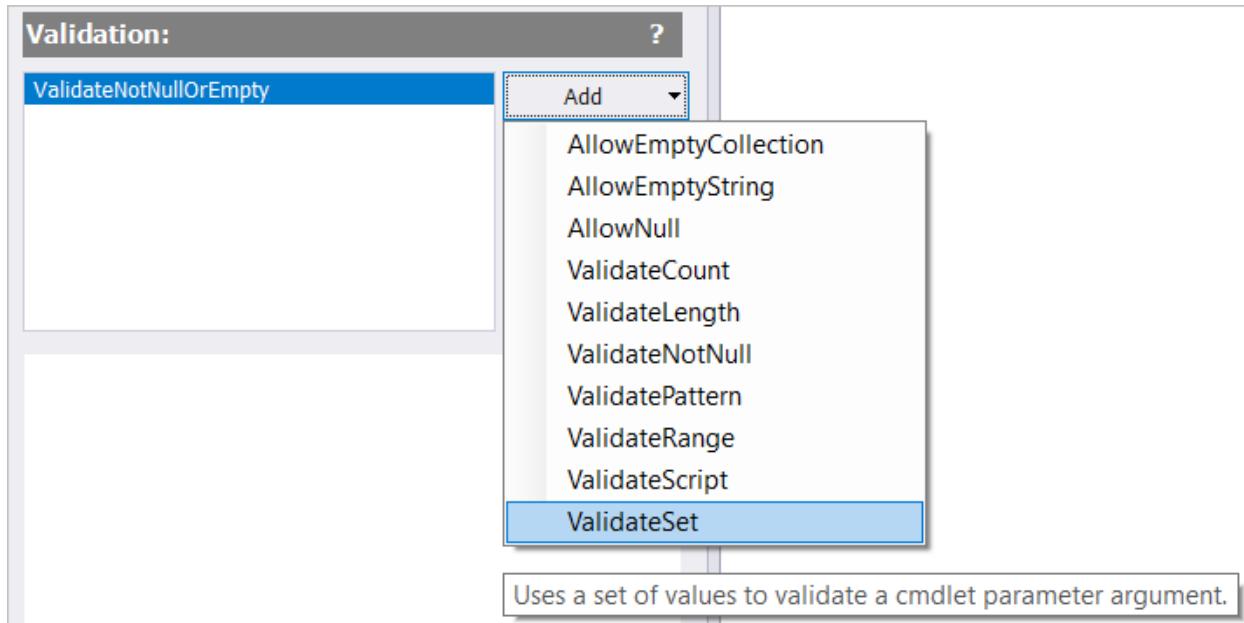
Provides an alternative name for the parameter (*optional*).

- **Default Value**

Specify a default value for the Parameter (e.g., \$env:ComputerName).

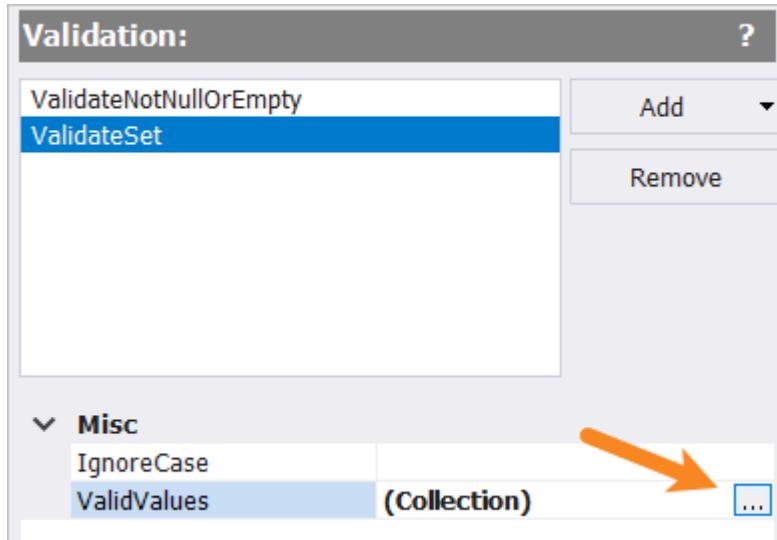
- **Validation**

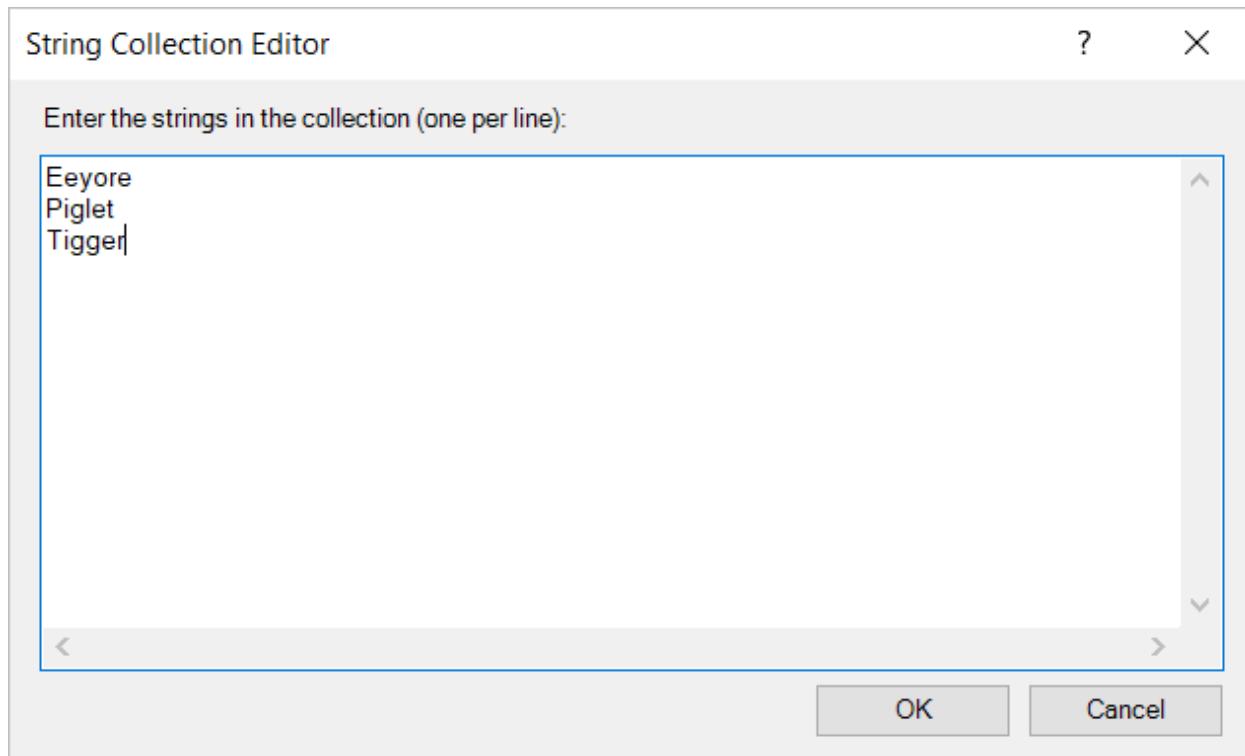
Use the Add drop-down to add validation attributes to the parameter:



Clicking a validation attribute will display a pop-up Help definition.

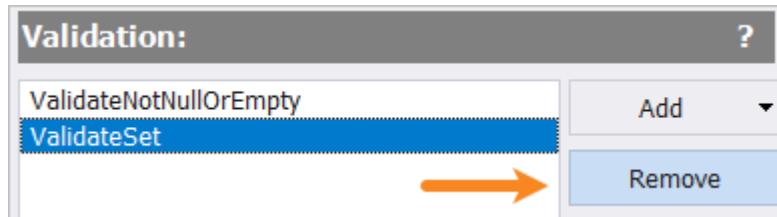
Once added, you can modify the validation attribute using the property grid:





## To remove a validation attribute

Select the validation attribute and click Remove:



Once you have added all of the parameters and entered the information required, press the **OK** button in the Function Builder dialog to generate the code in your script:

```
35     .PARAMETER Name
36         The name parameter for this cmdlet.
37
38     .EXAMPLE
39             PS C:\> Get-Data -Name Eeyore
40
41     .NOTES
42         Additional information about the function.
43     #>
44     function Get-Data
45     {
46         [CmdletBinding(DefaultParameterSetName = 'Name Set',
47                         ConfirmImpact = 'Medium',
48                         SupportsShouldProcess = $true)]
49         [OutputType([string])]
50         param
51         (
52             [Parameter(ParameterSetName = 'Name Set',
53                         Mandatory = $true,
54                         HelpMessage = 'Please enter the users name.')]
55             [ValidateNotNullOrEmpty()]
56             [ValidateSet('Eeyore', 'Piglet', 'Tigger')]
57             [Alias('nm')]
58             [string]$Name
59         )
59 }
```

### 5.11.1.10 Special Considerations

It is important to be aware of the following considerations when using the Function Builder:

#### Renaming Functions

When you rename an existing function within the Function Builder, PowerShell Studio will open the *Preview* dialog if there are any references outside of the function declaration, allowing you to select which of the function references you wish to update in the script.

#### Comment-Based Help

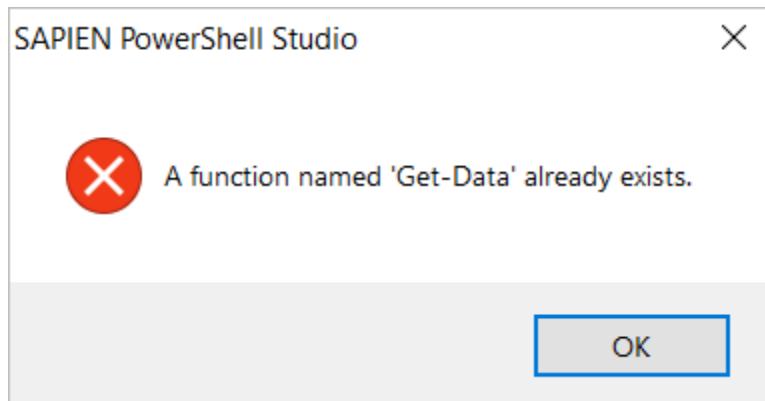
A [Comment-Based help](#) block will be inserted in your script only if you enter information in the Function Builder *Synopsis* or *Description* fields.

#### Comments in the Parameter Block

If the Function Builder encounters any comments in the function's parameter block, it will automatically assign the comment to the parameter's description—all parameter comments will be moved to the comment-based help block.

## Name Validation

The Function Builder will prevent duplicate functions by checking if the function name already exists in your script:



## Undo Changes

You can undo changes in the Function Builder at any time by using the keyboard shortcut *Ctrl+Z*.

### 5.11.2 Create Functions from Selection

When inserting a new function you can select a section of script and the Function Builder will use the selected text as the body of the function.

The Function Builder will pick potential parameters and display the following dialog where you can select the variables you want to convert into parameters:

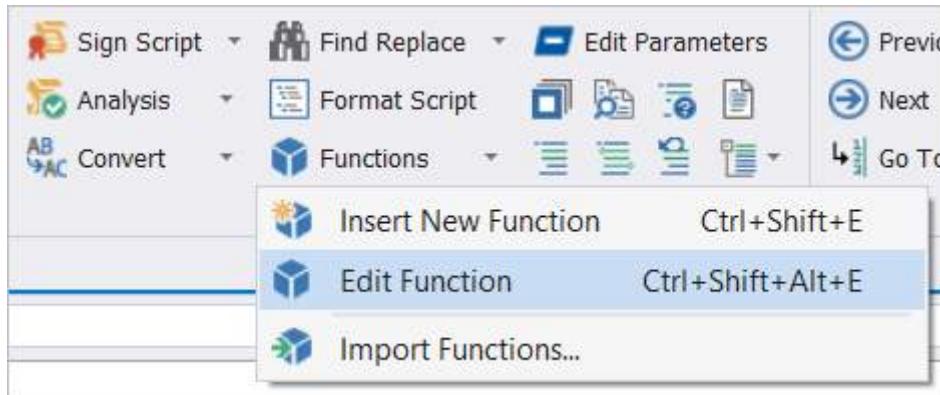
The screenshot shows the PowerShell Studio interface with the 'Disk Space Chart.ps1' script open in the code editor. A 'Convert to Parameters' dialog box is displayed over the script. The dialog lists variables found in the script: 'buttonSave\_Click', 'chart1', and 'Disks'. The 'Disks' variable is selected and highlighted with a blue border. The dialog includes 'OK', 'Cancel', 'Check All', and 'Uncheck All' buttons.

```
224 {
225
226     #Get Disk space using WMI and make sure it is an array
227
228     $Disks = @(Get-WMIObject Win32_LogicalDisk -filter "DriveType=3" )
229
230     #Remove all the current charts
231     Clear-Chart $chart1
232
233     #Loop through each dr
234     foreach($disk in $Dis
235     {
236         $UsedSpace =((($di
237         $FreeSpace = ($di
238
239         #Load a Chart for
240         Load-Chart $chart
241     }
242
243     #Set Custom Style
244     foreach ($Series in $S
245     {
246         $Series.CustomPro
247     }
248
249     $buttonSave_Click={
250         if($savefiledialog1.S
251     {
252         $Chart1.SaveImage($s
253     }
254 }
```

### 5.11.3 Editing Functions

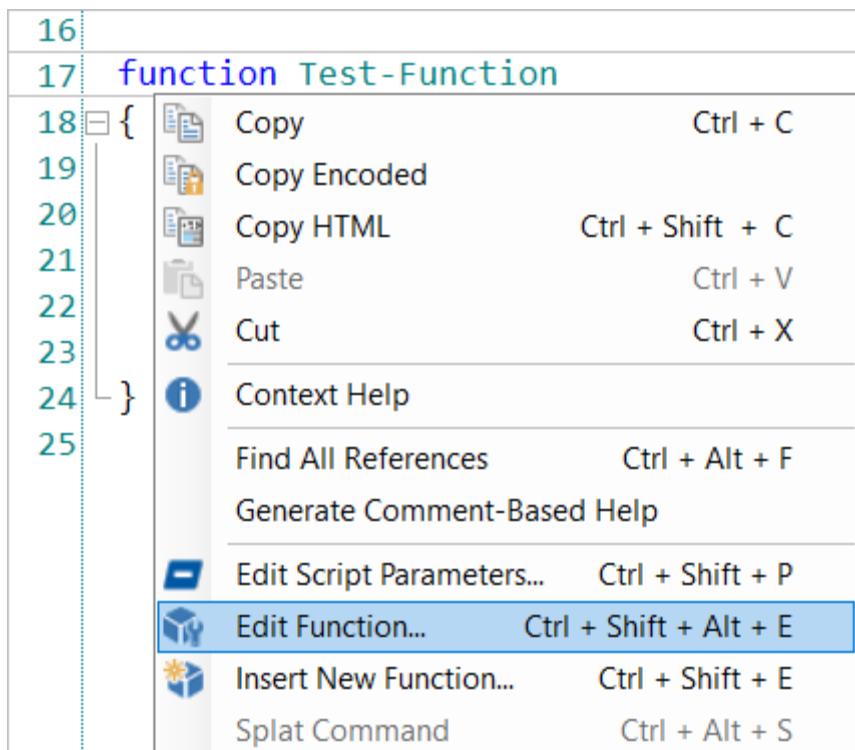
Use any of these options to edit an existing function:

- Position the caret on a function's declaration in the code editor, then on the **Home** tab > in the **Edit** section, click the **Functions** menu > select **Edit Function** (**Ctrl+Shift+Alt+E**):



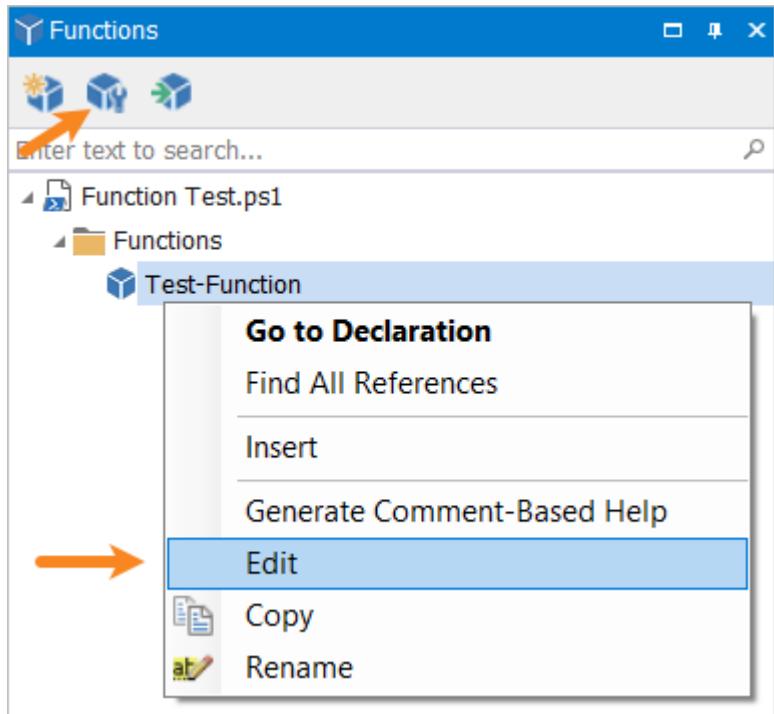
-OR-

- Right-click on a function's declaration in the code editor, and then select **Edit Function** (*Ctrl+Shift+Alt+E*):



-OR-

- In the Functions panel, select a function and then click the **Edit Function** button, or right-click on a function and select **Edit**:

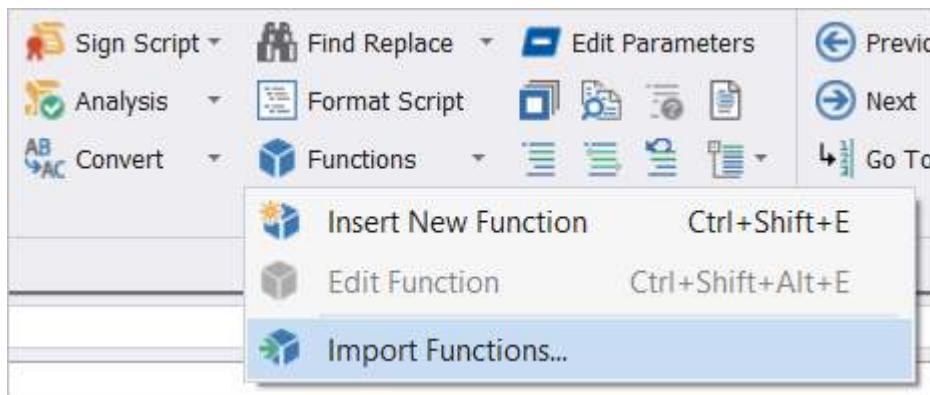


## 5.11.4 Importing Functions

PowerShell Studio allows you to import functions into your existing scripts or modules.

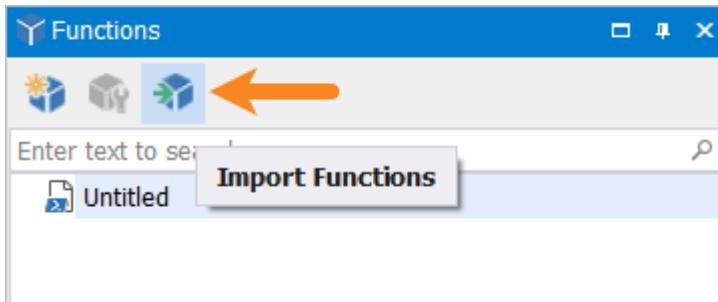
### To import functions

- On the Home tab > in the Edit section, select the Functions menu > then select Import Functions:

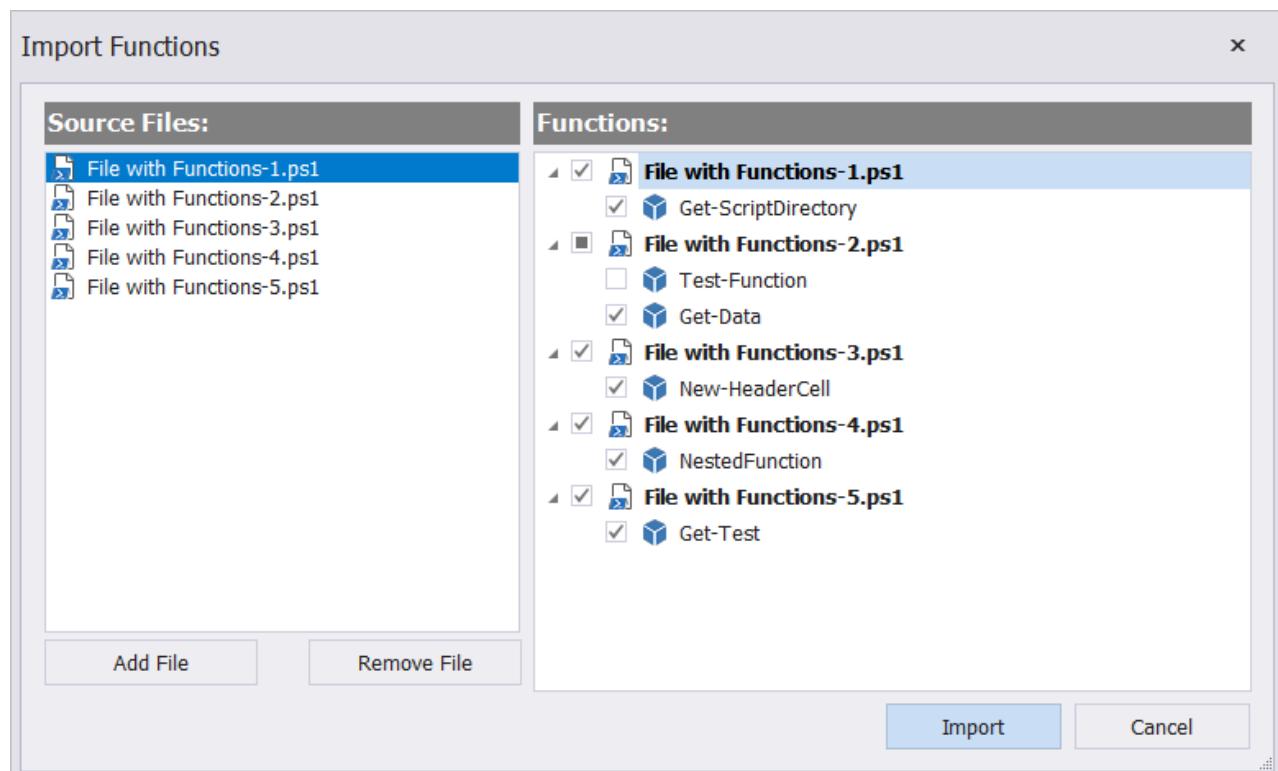


-OR-

- In the Functions panel, click the Import Functions button:



You will be prompted to select the files that contain the functions. You can select and remove files at any time. All functions in the selected files will be listed in the Import Functions dialog:



Select the functions that you want to import, and then click the **Import** button to insert the checked functions in the script:

```
16 #Sample function that provides the location of the script
17 function Get-ScriptDirectory
18 {
19 <#...
20     [OutputType([string])]
21     param ()
22     if ($null -ne $hostinvocation)
23     {
24         Split-Path $hostinvocation.MyCommand.path
25     }
26     else
27     {
28         Split-Path $script:MyInvocation.MyCommand.Path
29     }
30 }
31
32 <#...
33
34
35
36
37
38
39 }
40
41 <#...
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60 function Get-Data
61 {
```

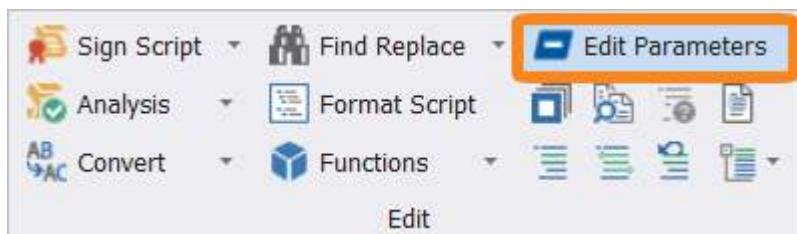
### 5.11.5 Parameter Builder

The Parameter Builder is a quick and easy way to insert a new parameter block, create a new parameter, or edit existing script parameters.

- ⓘ The Parameter Builder and the [Function Builder](#) share the same functionality, with the exception of the 'Name' portion of the builder.

#### To access the Parameter Builder

- From the Home tab > in the Edit section, select the **Edit Parameters** button (**Ctrl+Shift+P**):



-OR-

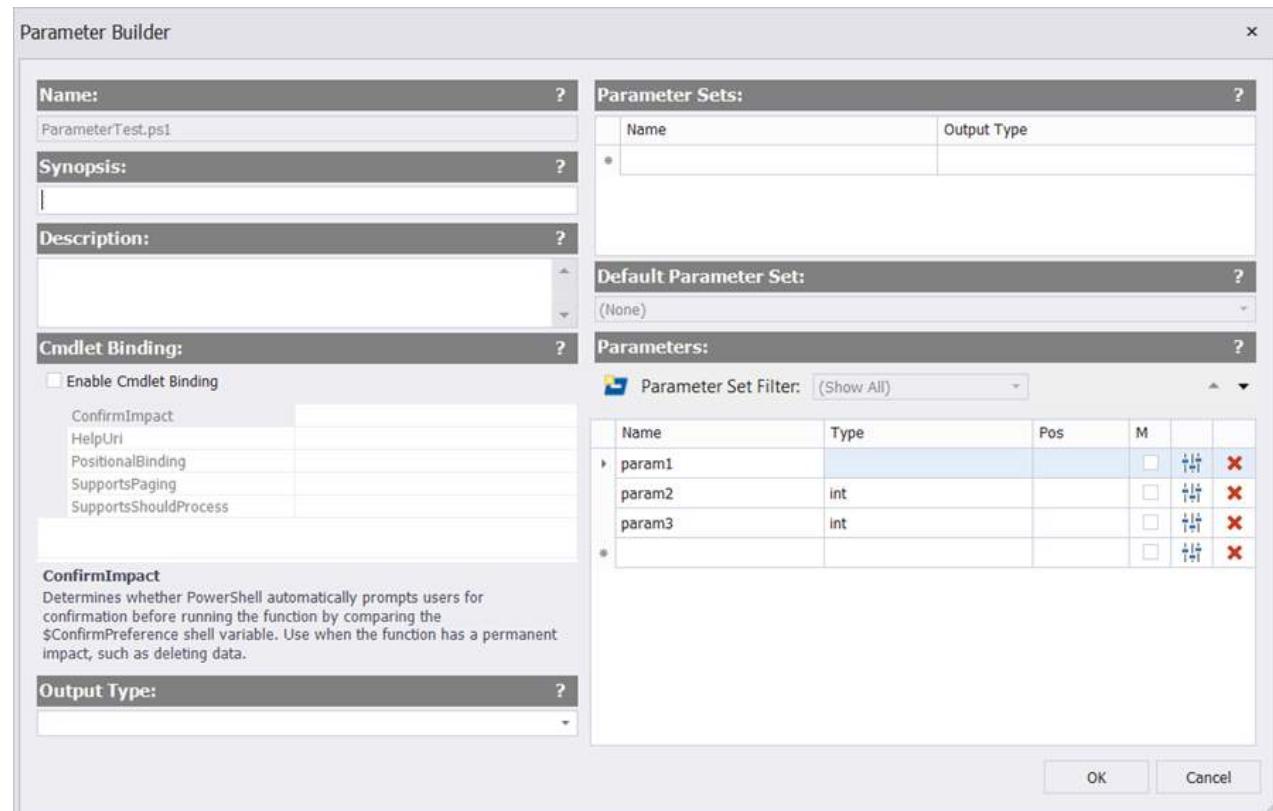
- Right-click in the editor and select **Edit Script Parameters...** (**Ctrl+Shift+P**):

```

9
10 #parameterTest.exe
11 param($param1 = 'd')
12
13 $EngineArgs
14 "Param1: $param1"
15 "Param2: $param2"
16 "Param3: $param3"
17
18 #
19 #$testing = "&"
20 #Add-PSSnapin Xens

```

You can use the Parameter Builder to specify parameters, parameter sets, validation, and help comments:



When you are done editing the parameters, click **OK** and PowerShell Studio will produce a parameter block for your script.

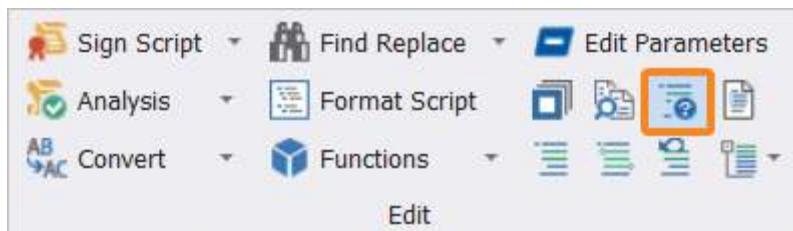
## 5.12 Comment-Based Help

PowerShell Studio can automatically generate comment blocks for existing functions. This section explains how to generate comment-based help, and shows you how to use comment-based help templates.

### Generating Comment-Based Help

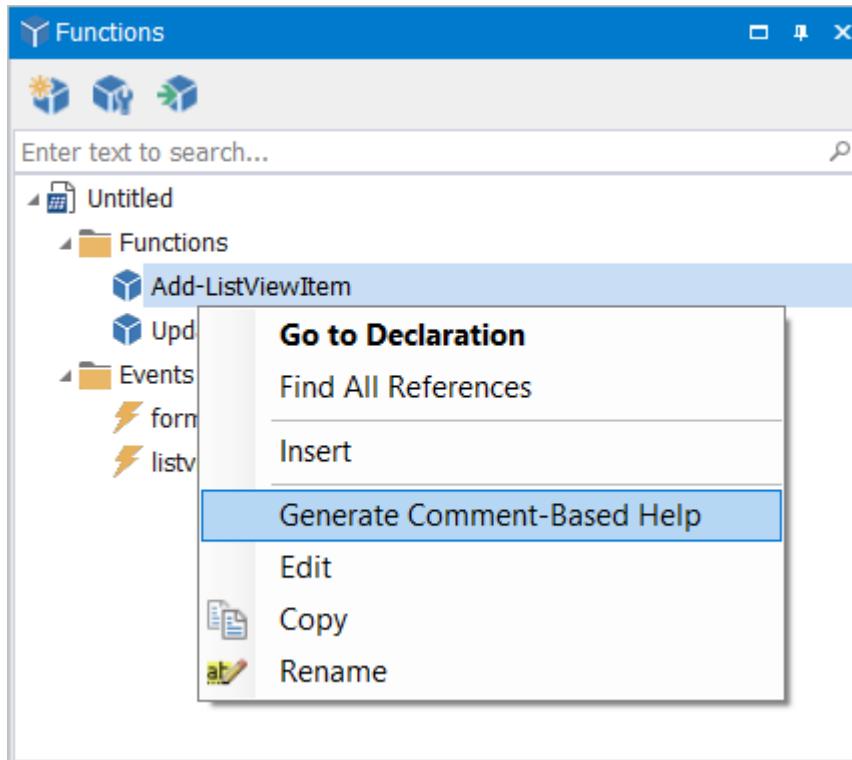
Use any of these options to generate comment blocks for an existing function:

- Position the cursor on the function's declaration in the code editor, and then on the **Home** tab > in the **Edit** section, select the **Generate Comment-Based Help** button:



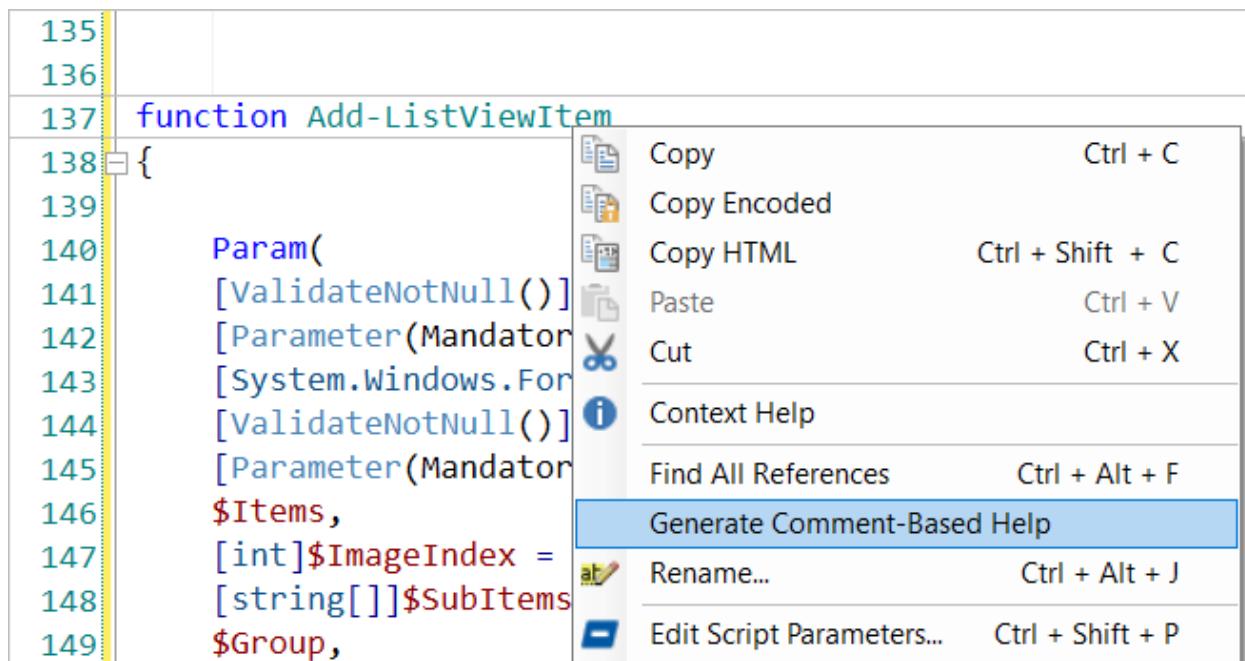
-OR-

- In the Functions panel, right-click on a function and select **Generate Comment-Based Help**:



-OR-

- Right-click on the function's declaration in the code editor, and then select **Generate Comment-Based Help**:



PowerShell Studio will insert a help comment tailored to the specific function:

```
135
136
137 <#
138     .SYNOPSIS
139         A brief description of the Add-ListViewItem function.
140
141     .DESCRIPTION
142         A detailed description of the Add-ListViewItem function.
143
144     .PARAMETER ListView
145         A description of the ListView parameter.
146
147     .PARAMETER Items
148         A description of the Items parameter.
149
150     .PARAMETER ImageIndex
151         A description of the ImageIndex parameter.
152
153     .PARAMETER SubItems
154         A description of the SubItems parameter.
155
156     .PARAMETER Group
157         A description of the Group parameter.
158
159     .PARAMETER Clear
160         A description of the Clear parameter.
161
162     .EXAMPLE
163         PS C:\> Add-ListViewItem -ListView $ListView -Items $Items
164
165     .NOTES
166         Additional information about the function.
167 #>
168 function Add-ListViewItem
169 {
170     Param(
```

The help structure is inserted into the script, and you will need to add the descriptive text and edit as appropriate.

By default, the help comment is inserted *before* the function declaration (as shown above).

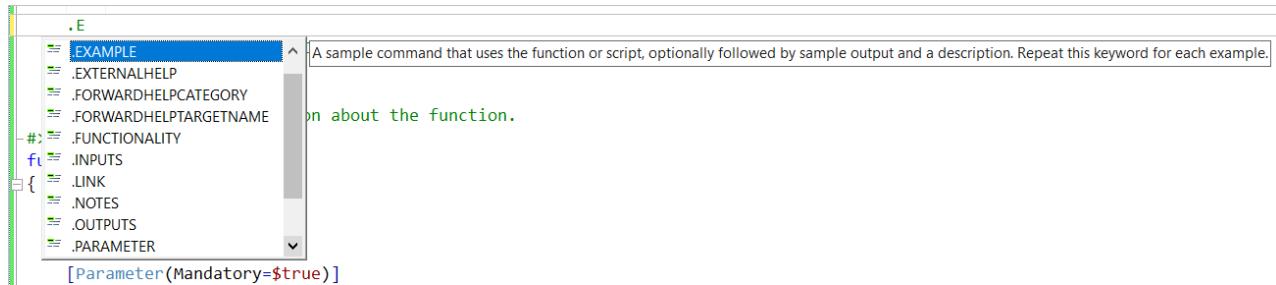
👉 The help comment can be inserted *inside* the function declaration by selecting Home > Options > Editor > Editor Settings > Insert comment-based help *Inside Function*:

```
137 function Add-ListViewItem
138 {
139 <#
140     .SYNOPSIS
141         A brief description of the Add-ListViewItem function.
142
143     .DESCRIPTION
144         A detailed description of the Add-ListViewItem function.
145
146     .PARAMETER ListView
147         A description of the ListView parameter.
148
149     .PARAMETER Items
150         A description of the Items parameter.
151
152     .PARAMETER ImageIndex
153         A description of the ImageIndex parameter.
154
155     .PARAMETER SubItems
156         A description of the SubItems parameter.
157
158     .PARAMETER Group
159         A description of the Group parameter.
160
161     .PARAMETER Clear
162         A description of the Clear parameter.
163
164     .EXAMPLE
165         PS C:\> Add-ListViewItem -ListView $ListView -Items $Items
166
167     .NOTES
168         Additional information about the function.
169 #>
170
171     Param(
```

## Displaying Help Keyword Information

PowerShell Studio's PrimalSense can provide assistance when you are editing comment-based help.

Within the comment help block, type a ". ." (dot) to trigger the help keyword menu. Scroll up or down in the menu to highlight a specific keyword and display the keyword information. Press < Enter > to insert the highlighted keyword into the comment block:



## 5.12.1 Comment-Based Help Templates

PowerShell Studio features the ability to create templates for the [Generate Comment-Based Help](#) feature.

### 5.12.1.1 About the Comment-Based Help Template

The predefined comment-based help template contains the following:

```
1 1<#
2 .SYNOPSIS
3     A brief description of the %NAME% %TARGETTYPE%.
4 .DESCRIPTION
5     A detailed description of the %NAME% %TARGETTYPE%.
6 .PARAMETER
7     A description of the %PARAMETER% parameter.
8 .EXAMPLE
9     %EXAMPLE%
10 .OUTPUTS
11     %OUTPUTS%
12 .NOTES
13     Additional information about the %TARGETTYPE%.
14 #>
15
```

The template file contains the typical comment-based help block with some token variables and a single parameter section. When the user generates comment-based help for a function or file, PowerShell Studio will read the template file. Depending on if the function already has comment-based help, it will behave as follows:

If comment-based help is not present	PowerShell Studio will insert all of the sections in order, as defined by the template.
If comment-based help is present	PowerShell Studio will check whether each section within the template is present in the existing comment and insert if necessary. In addition, the existing comment's sections will be reordered to match that of the template's.

### Parameter Section

---

The comment-based help template contains a single parameter section that defines each new parameter's text. Notice that there is no parameter name after .PARAMETER. When PowerShell Studio generates the .PARAMETER section in the template, it will place all of the function's parameters sequentially within the comment block.

For example, all of the parameters shown in the template above will be inserted sequentially after the .DESCRIPTION section and before the .EXAMPLE section.

### Empty Sections

---

Empty sections will not be inserted. For example, if the .OUTPUTS section in the template above is empty after replacing the variables (the function has no output), PowerShell Studio will not include it in the generated comment.

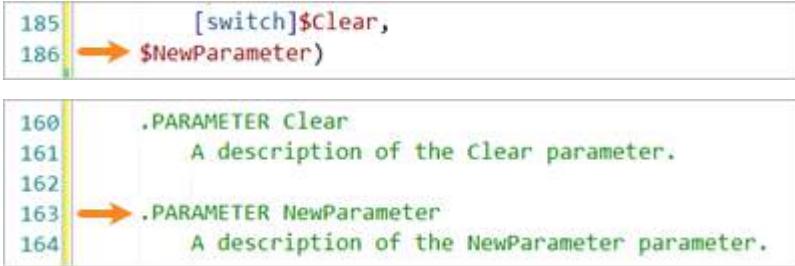
#### 5.12.1.2 Comment-Based Help Template Variables

When a comment block is inserted in a script, PowerShell Studio dynamically expands the following template variables:

%NAME%	Inserts the name of the function or file.
%TARGETTYPE%	Inserts "function" if the comment is for a function. Inserts "file" if the comment is for a file.
%HELPURL%	Inserts the Help URL defined in the function's Cmdlet Binding attribute.
%PARAMETER%	Inserts the parameter name (Only valid within the Parameter section).
%EXAMPLE%	Inserts a generated example using the existing parameters.
%OUTPUTS%	Inserts a list of outputs determined by the Output attribute.

 For details about template variables and formatting, see [File Type Templates > Template Variables](#) (126)

 If you add a new parameter to an existing function, simply **Generate Comment-Based Help** content again and PowerShell Studio will append any missing parameters to the comment block:



```

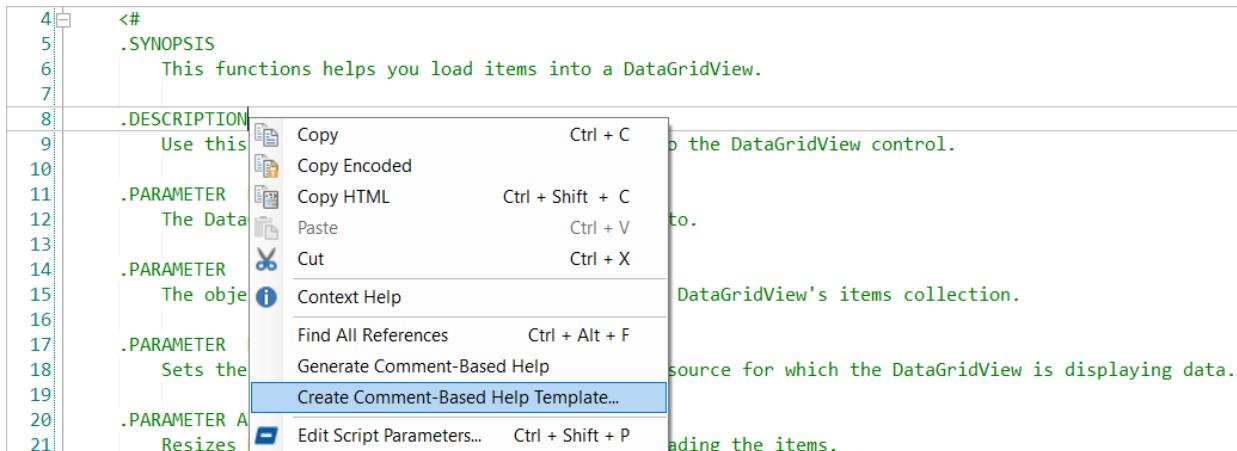
185     [switch]$clear,
186     $NewParameter)
187
188
189
190     .PARAMETER Clear
191         A description of the clear parameter.
192
193     .PARAMETER NewParameter
194         A description of the NewParameter parameter.
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360

```

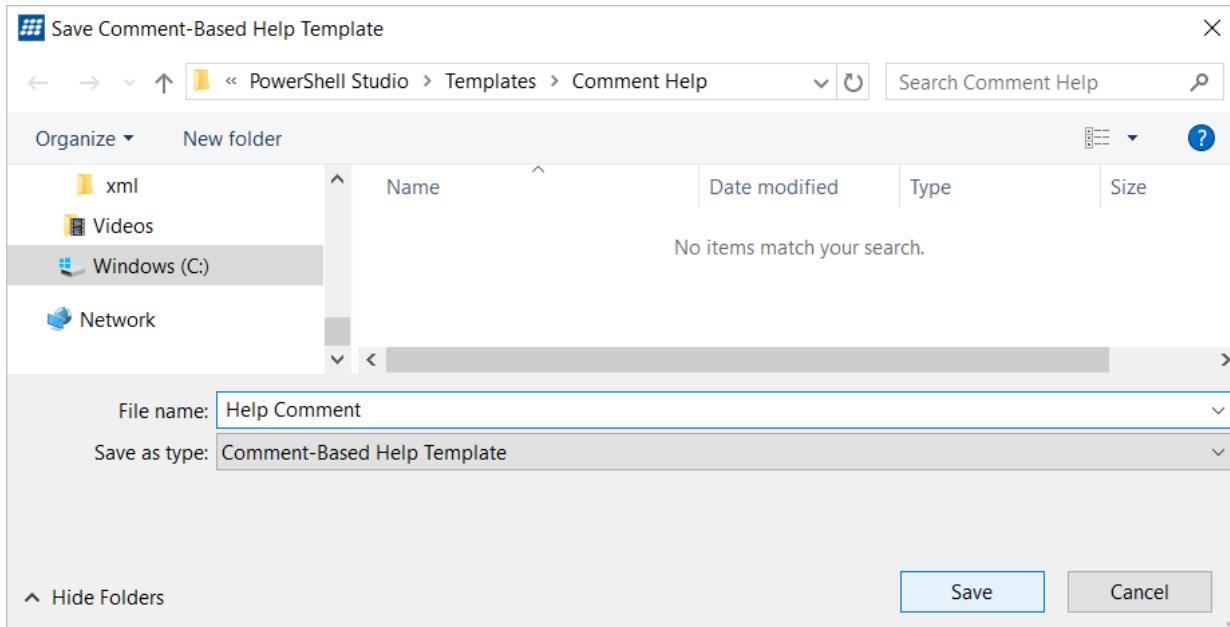
### 5.12.1.3 Creating a Comment-Based Help Template

#### To create a comment-based help template

1. Right-click on the comment-based help you wish to convert into a template and select the **Create Comment-Based Help Template** option from the context menu:



2. In the Save Comment-Based Help Template dialog, enter a file name and then click **Save**:



The resulting template file will be saved as a ps1 file in the following folder:

`%AppData%\Roaming\SAPIEN\PowerShell Studio\Templates\Comment Help\`

3. Edit the comment file to include the necessary sections, variables, and text. If the original comment-based help has one or more parameter sections, PowerShell Studio will automatically format the .PARAMETER section for the template:

```
1  <#
2   .SYNOPSIS
3     This functions helps you load items into a DataGridView.
4
5   .DESCRIPTION
6     Use this function to dynamically load items into the DataGridView control.
7
8   .PARAMETER
9     The %PARAMETER% control you want to add items to.
10  #>
```

4. Save the file by clicking **File > Save (Ctrl+S)**.

**i** You can generate a new template from an existing template by using the predefined template as a starting point and following the same procedure described above. PowerShell Studio's predefined templates are located in the following folder:

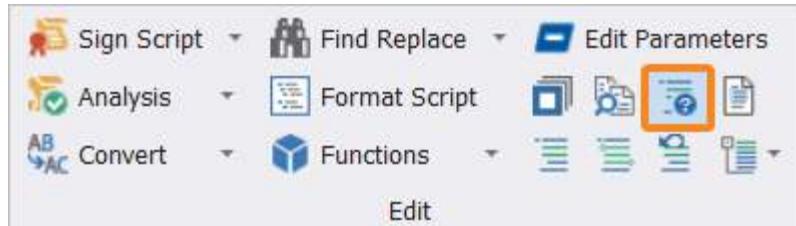
`%ProgramData%\SAPIEN\PowerShell Studio <yyyy>\Templates\Comment Help`

#### 5.12.1.4 Selecting an Existing Comment-Based Help Template

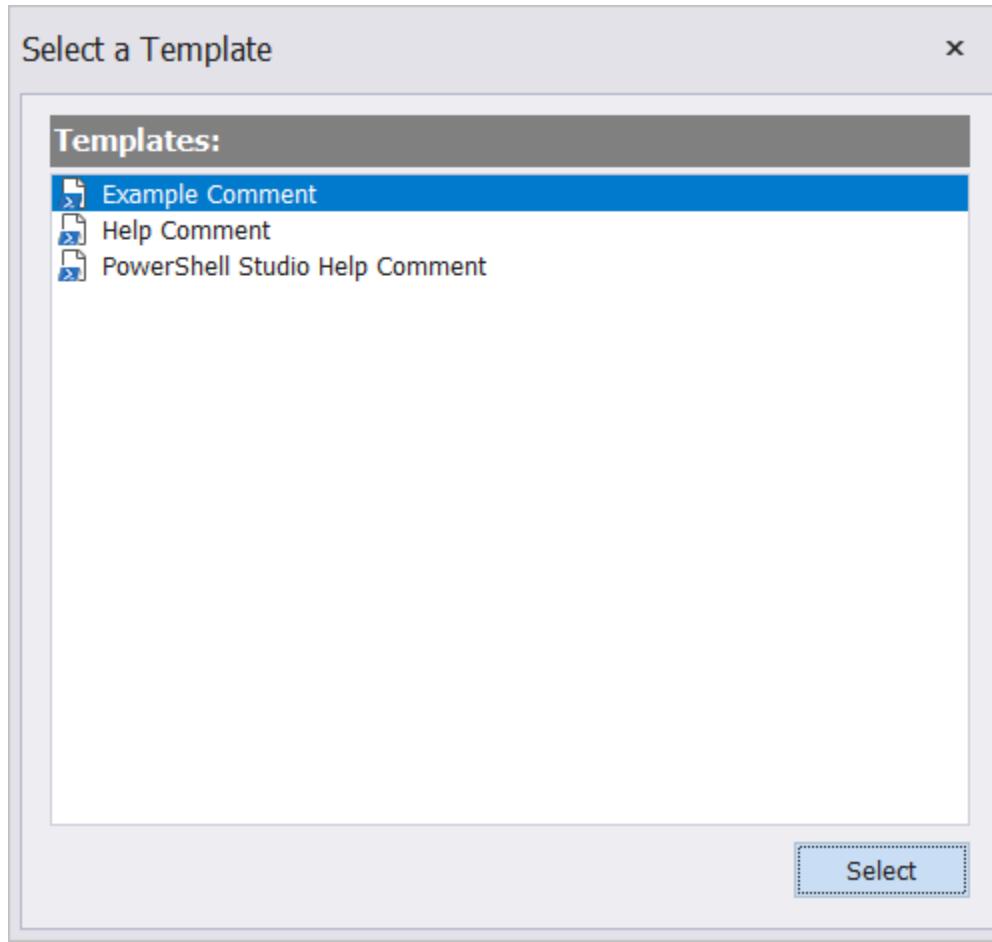
PowerShell Studio allows you to define multiple templates, which allows you to insert predefined comment-based help as needed.

## To insert the contents of a comment help template into your script

Position the caret in the function declaration and then on the **Home** tab > in the **Edit** section, click the **Generate Comment-Based Help** button:



If more than one template exists, PowerShell Studio will present you with the following selection dialog:



Highlight the desired template to be applied, and then click Select.

### 5.12.1.5 Multi-line or Single-line Comments

The [comment-based help templates](#)<sup>114</sup> support either single-line comments or multi-line comments.

If you prefer to use single-line comments you can create a template using single-line comments and, when applied, any existing multi-line comments will be converted to single-line. If you have a multi-line comment template, it will convert the existing single-line comments to a multi-line comment.

- A template's formatting is not taken into account when generating comment-based help.

## 5.13 File Type Templates

PowerShell Studio provides templates for various file types, and also allows you to create new templates.

### 5.13.1 Using Predefined File Templates

This topic provides a list of the available predefined file templates and shows you how to create a script using a predefined template.

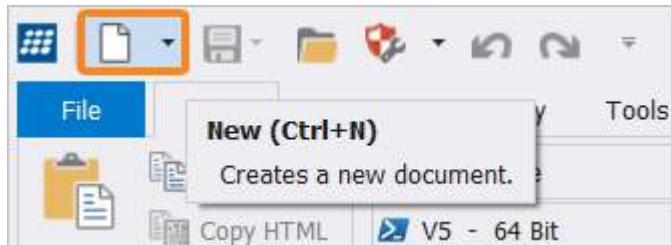
## Predefined Templates

Template Type	File Extension	Description
PowerShell Script	ps1	Creates a PowerShell script file.
PowerShell Form	psf	Creates an empty PowerShell form.  For information on creating Form or Grid templates, see <a href="#">GUI Designer &gt; Form Templates</a> <small>[171]</small> .
Module Manifest	psd1	Creates a PowerShell module manifest file.
Module Script	psm1	Creates a PowerShell module script file.
PowerShell Service Script	ps1	Creates a PowerShell service script. Used for the service packaging engine.
PowerShell Class	ps1	Creates a PowerShell script file with a class declaration. When you use this template, PowerShell Studio immediately allows you to rename the class directly within the editor. <i>If you add the class template as part of a project, it will use the specified file name as the default name of the class.</i>
C# File	C# File.cs	Creates an empty C# file. You can use this template as a quick way of writing C# code for the Add-Type cmdlet.
Text File	Text File.txt	Creates an empty text file.

## Using Predefined File Templates

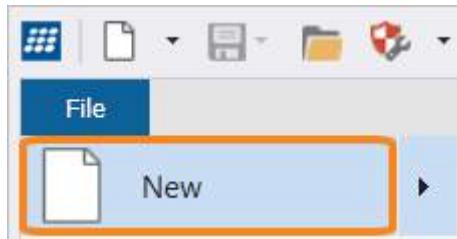
### To create a script from a predefined template

- On the Quick Access menu, select New (*Ctrl+N*):



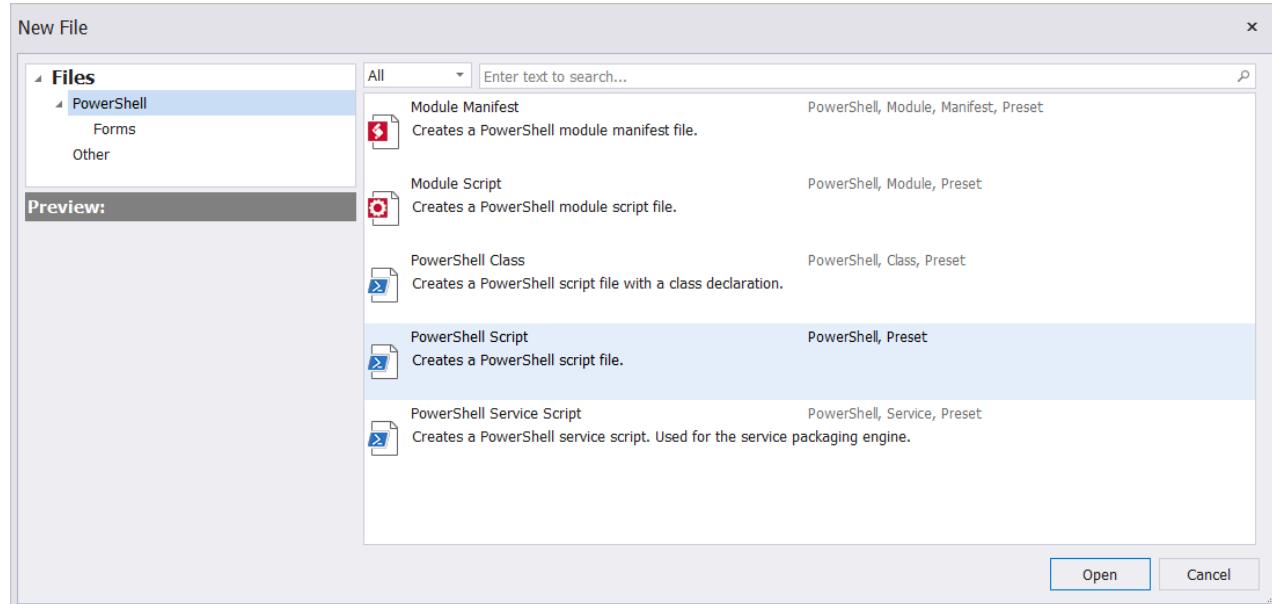
-OR-

- On the File menu, select New (*Ctrl+N*):



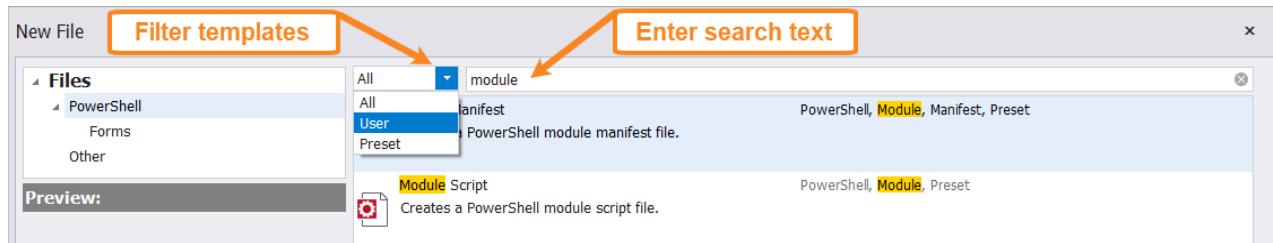
👉 If you know what template you want to use, you can select it directly from the **Quick Access** menu or the **File > New** menu.

Browse through the template categories in the New File dialog, then select a template and click **Open**:



PowerShell Studio will open the template file in the script editor.

👉 To quickly locate a particular template, use the drop-down menu to filter the templates by User defined or Preset, or enter a term to search by tag/keyword:



## 5.13.2 Creating New File Templates

If the predefined templates in PowerShell Studio do not meet your needs, you can create your own.

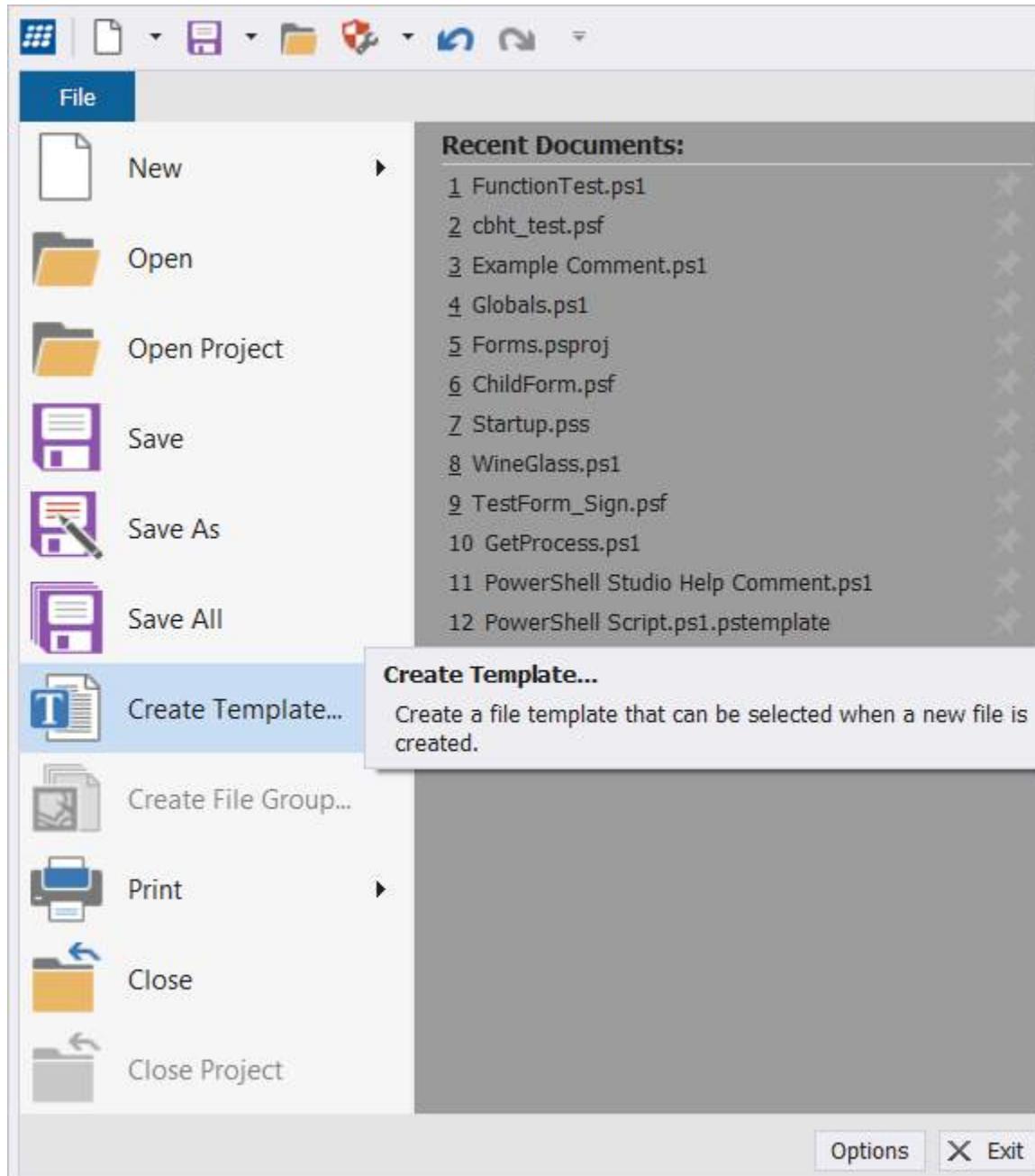
### Creating New File Templates

---

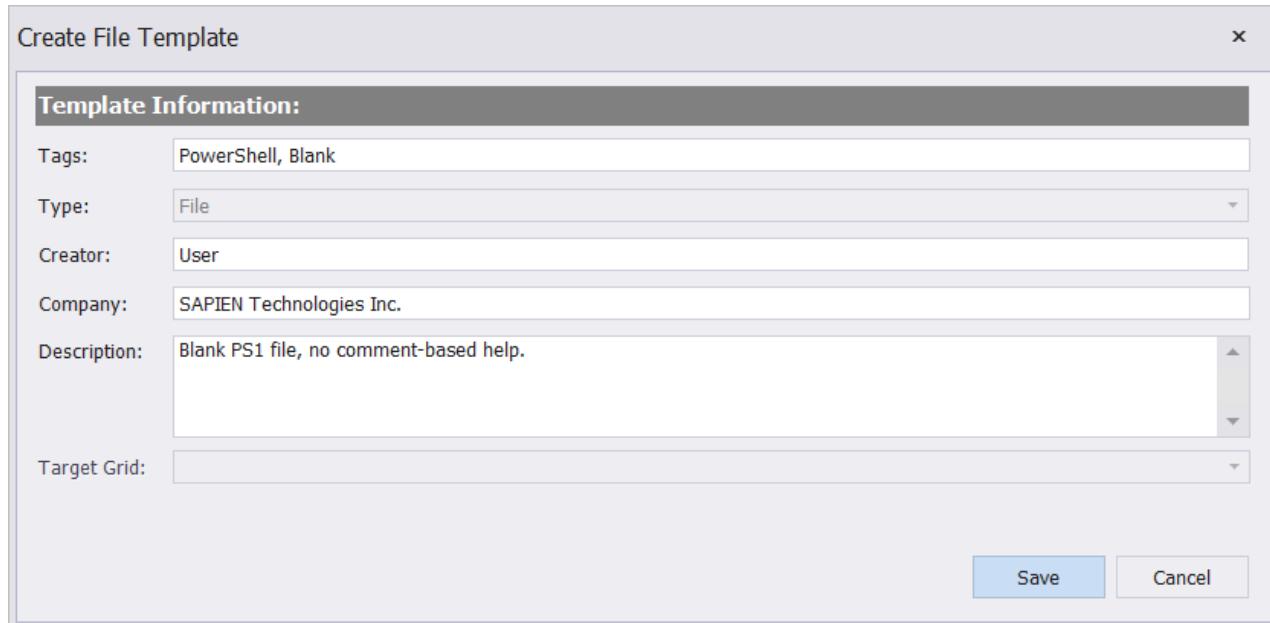
#### To create a template

Create a new file or open an existing file or template.

Save the file, and then on the **File** menu > select **Create Template**:



The Create File Template dialog will open:



Fill in the template properties and then click **Save**:

- **Tags**

PowerShell Studio will automatically set some default tags based on the file extension. You can add additional comma-separated tags.

- **Type**

PowerShell Studio supports two types of templates:

- **File (form)**

This template is used for new files.

- **Grid**

This option is used when exporting a GUI from the Database Browser or the WMI Browser.

**i** To select a Grid template, the psf file must contain a DataGridView control.

- **Creator**

Your name. (The value for this field is taken from **Home > Options > General > Username**).

- **Company**

Your company details. (The value for this field is taken from **Home > Options > General > Organization**).

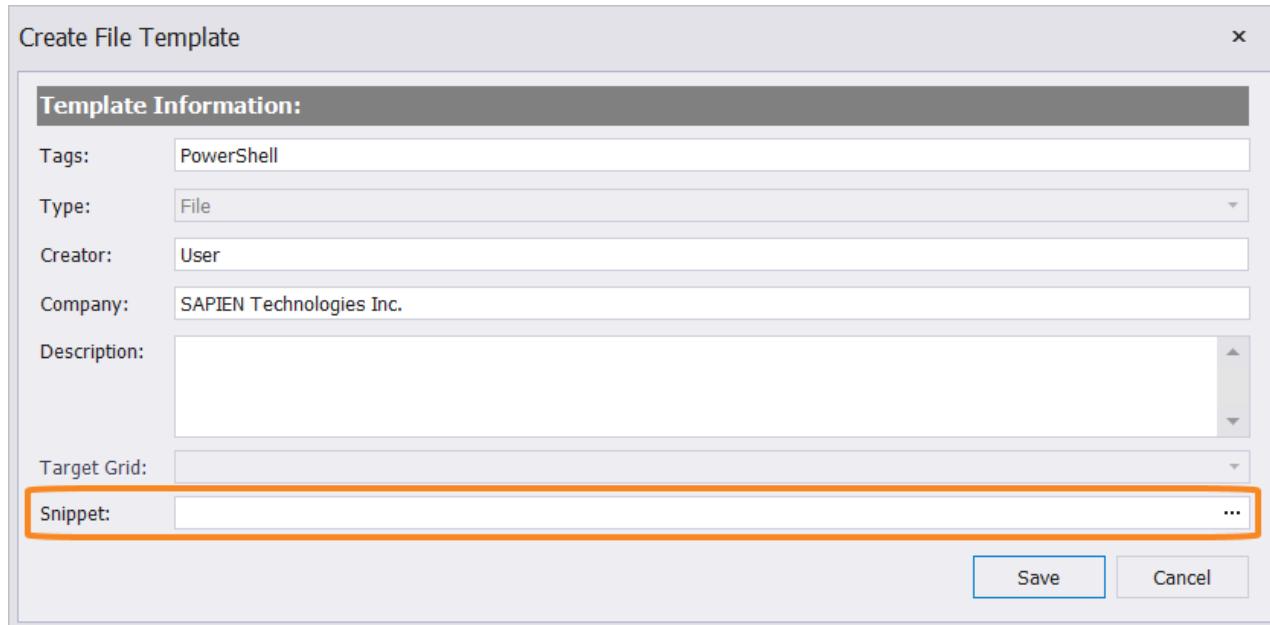
- **Description**

Description of the template purpose.

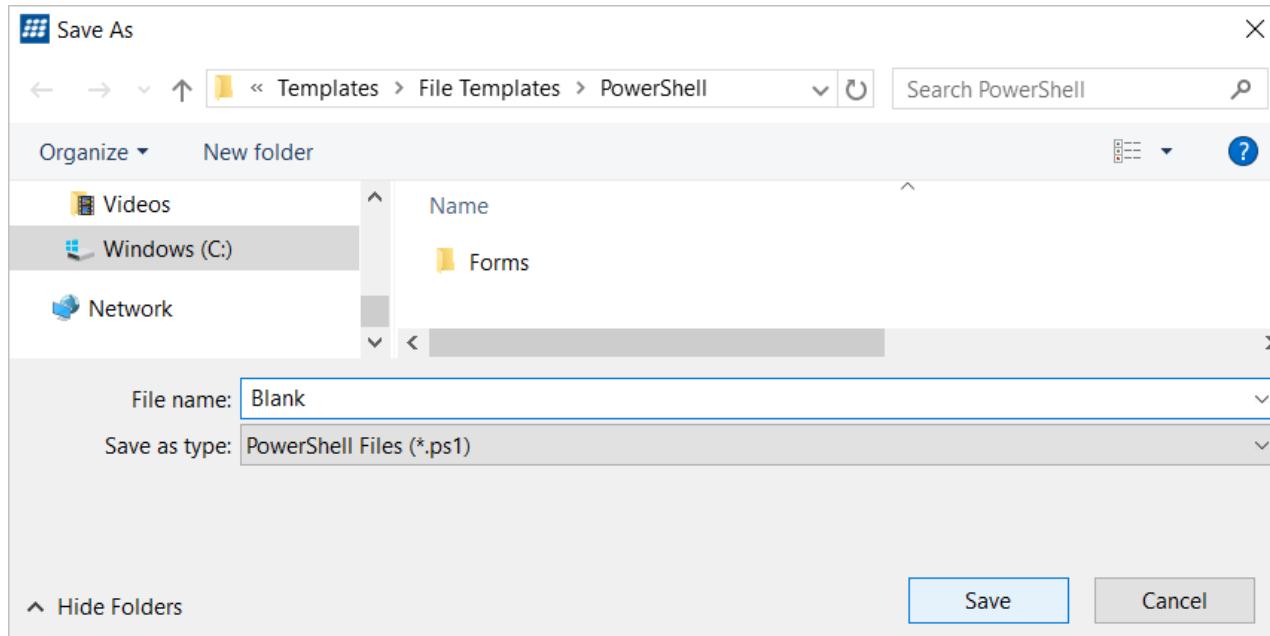
- **Target Grid**

This option is available if you are creating a grid template.

If you specified the %SNIPPET% variable in your script, the Create File Template dialog will include a Snippet field where you can select the desired snippet to trigger when the template is loaded. The selected snippet will be included with the template:

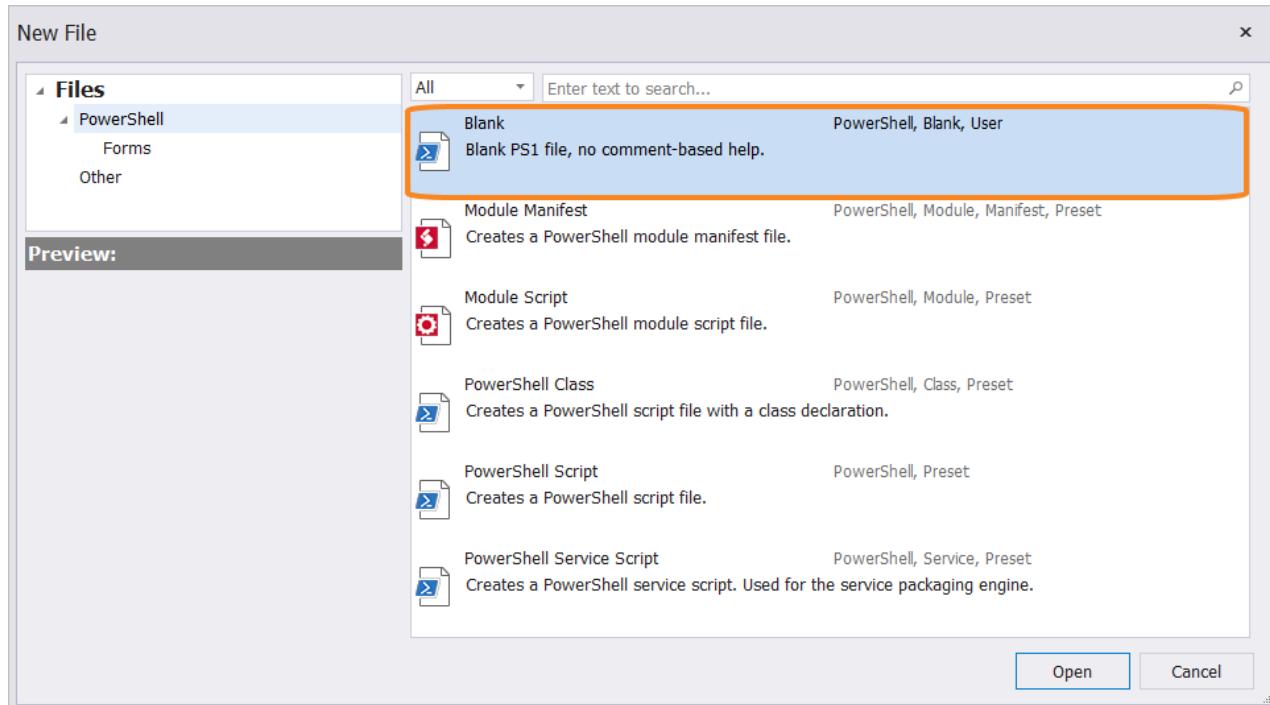


Enter a name for the template in the Save As dialog, and then click **Save**:



- i** The file type must have the same extension as the file type you wish the template to be applied to. For example, a file named UserTemplate.ps1 will only apply to .ps1 script files.

The new template will appear whenever you create a new file:



## Template Categories

Templates can be organized by categories (folders) to make it easier to browse your existing templates. When you save a template in the Save As dialog (shown above), the folder that you choose to save the template to will determine the category.

The default location for user created templates is %AppData%\SAPIEN\PowerShell Studio\Templates\File Templates\PowerShell.

**PowerShell templates are stored in a PowerShell folder**

[Template Directory]\File Templates\PowerShell\

**PowerShell GUI forms (psf) are stored in a Forms folder**

[Template Directory]\File Templates\PowerShell\Forms

The default location can be changed in Home > Options > General > Directories > Template Directory.

👉 You can create new categories by simply creating a new folder when saving your template.

### 5.13.3 Template Variables

PowerShell Studio supports template variables that automatically expand when a template is loaded.

## Template Variables

%AppName%	PowerShell Studio name.
%AppVersion%	PowerShell Studio version.
%UserName%	The user name specified in the settings.
%Company%	The user company name specified in the settings.
%Year%	Current Year <yyyy>.
%Date%	Current Date <m/dd/yyyy>.
%Time%	Current Time <h:mm PM>.
%FileName%	Inserts the file's name. This will be empty for new non-project files.
%FileTitle%	Inserts the file name without the extension.
%ProjectName%	Inserts the file's project name.
%GUID%	Inserts a unique GUID.
%SNIPPET%	Inserts the snippet that comes with the template script. This is handled automatically when creating a template. <i>Templates only trigger the first %SNIPPET% variable—subsequent uses of the variable will be ignored.</i>
%SNIPPET:SHORTCUT%	Inserts the snippet with the matching shortcut.  <i>This snippet must be present in the Snippet Panel and is not included with the template.</i> <i>For example: %SNIPPET:MSGBOX% will insert the message box snippet when the template is first loaded.</i>

## Template Variable Formatting

When using variables in your templates, it may be necessary to format the value for particular circumstances. For example, the template variables in PowerShell Manifest (psd1) are often contained

in quotes, therefore the value must be formatted/encoded to prevent it from breaking the existing string.

To encode the value, use the following format:

**%VARIABLE:FORMAT%**

Where VARIABLE is the name of the variable and FORMAT is the name of the formatting type.

<i>Format Name</i>	<i>Description</i>
PSSingle	Format for a PowerShell single-quoted string.
PSDouble	Format for a PowerShell double-quoted string.
C	Format for a C/C++ string.
HTML	Format for an HTML string.
Object	Format for object names (class/variable/members).

For example, use the following to format USERNAME into a single-quote PowerShell string:

**%USERNAME:PSSingle%**

If you want the variable to be a name for an object, such as a class name, then use the Object format:

**%FILETITLE:Object%**

**i** In some cases, variables may return an empty string and this may be undesirable, especially when the variable is used for an object's name. The template format allows you to set defaults to handle cases like these.

## 5.14 Rename Refactoring

Rename refactoring allows you to rename and update object references throughout the script for variables, controls, parameters, functions, events, class names, class member names, and more. Rename refactoring also provides a preview so you can quickly see where the object is used and selectively decide if you want to proceed with any change.

### How to Rename Objects

---

#### To initiate Rename Refactoring

---

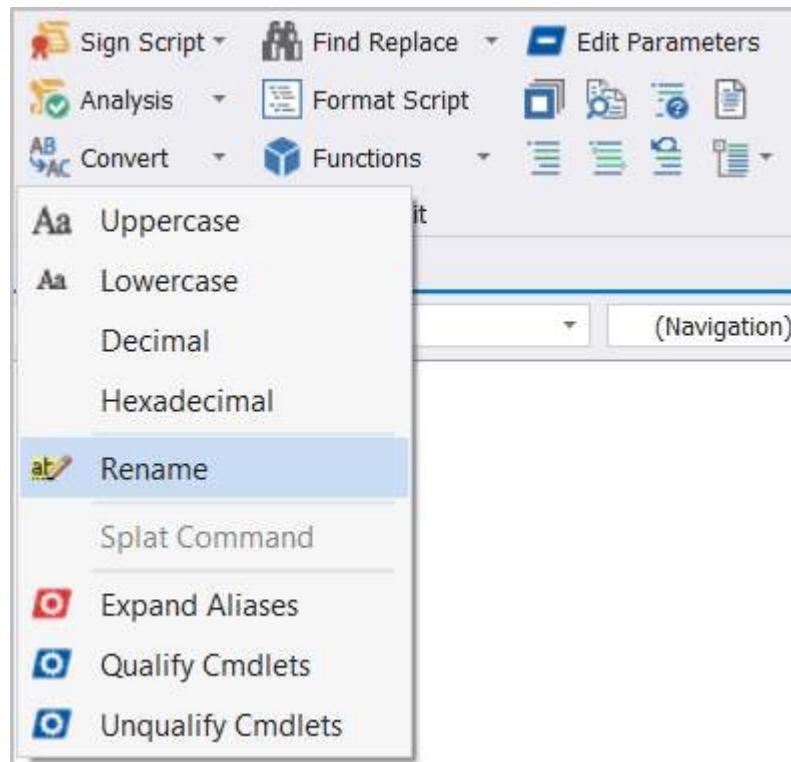
- Highlight the object you wish to rename, then right-click and select **Rename (Ctrl+Alt+J)**:

A screenshot of the PowerShell Studio interface. In the center, there is a code editor window displaying a PowerShell script. The script includes lines like 'foreach(\$disk in \$Disks)', '\$UsedSpace =(((\$disk.size - \$disk.freespace)/1', and '#Set Custom S'. A context menu is open over the variable '\$UsedSpace' at line 236. The menu contains several options: Copy (Ctrl + C), Copy Encoded, Copy HTML (Ctrl + Shift + C), Paste (Ctrl + V), Cut (Ctrl + X), Context Help, Find All References (Ctrl + Alt + F), Generate Comment-Based Help, and Rename... (Ctrl + Alt + J). The 'Rename...' option is highlighted.

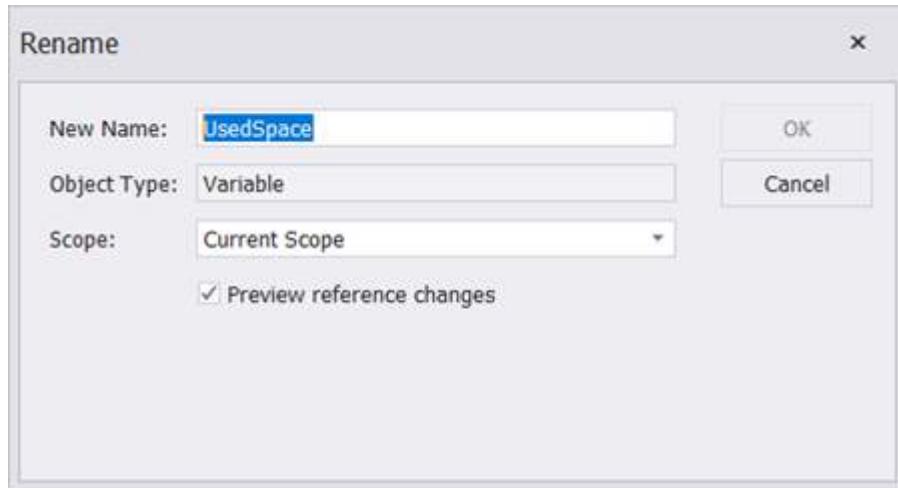
```
234 foreach($disk in $Disks)
235 {
236     $UsedSpace =((($disk.size - $disk.freespace)/1
237     [variable]$UsedSpace
238     #Load a C
239     Load-Char
240 }
241
242 #Set Custom S
243 foreach ($Ser
244 {
245     $Series.C
246 }
247 }
```

-OR-

- Highlight the object you wish to rename, then on the Home tab > in the Edit section, click the Convert drop-down > select Rename:



The Rename dialog will open:



Make your selections in the [Rename dialog](#), and then select **OK**.

## Rename dialog field descriptions

---

- **New Name**

Enter the new name for the selected object.

- **Object Type**

Displays the type of object that you are renaming. Different settings will be updated, depending on the type of object being renamed:

<b>Enum</b>	Updates the enumerator value declared in the script.
<b>Event</b>	Updates the control event variable and references in the designer.
<b>Function</b>	Updates the function declaration and function calls.
<b>GUI Control</b>	Updates the GUI control variable and updates any event name and return variables for projects.
<b>Method</b>	Updates the class methods and the class method references.
<b>Parameter</b>	Updates the parameter for the function and any calls using the parameter.
<b>Property</b>	Updates the class property and the class property references.
<b>Variable</b>	Updates variables within the document that aren't a parameter or GUI control.

- **Scope**

Select the scope for the objects that will be replaced. The list of available scopes will vary depending on the object type:

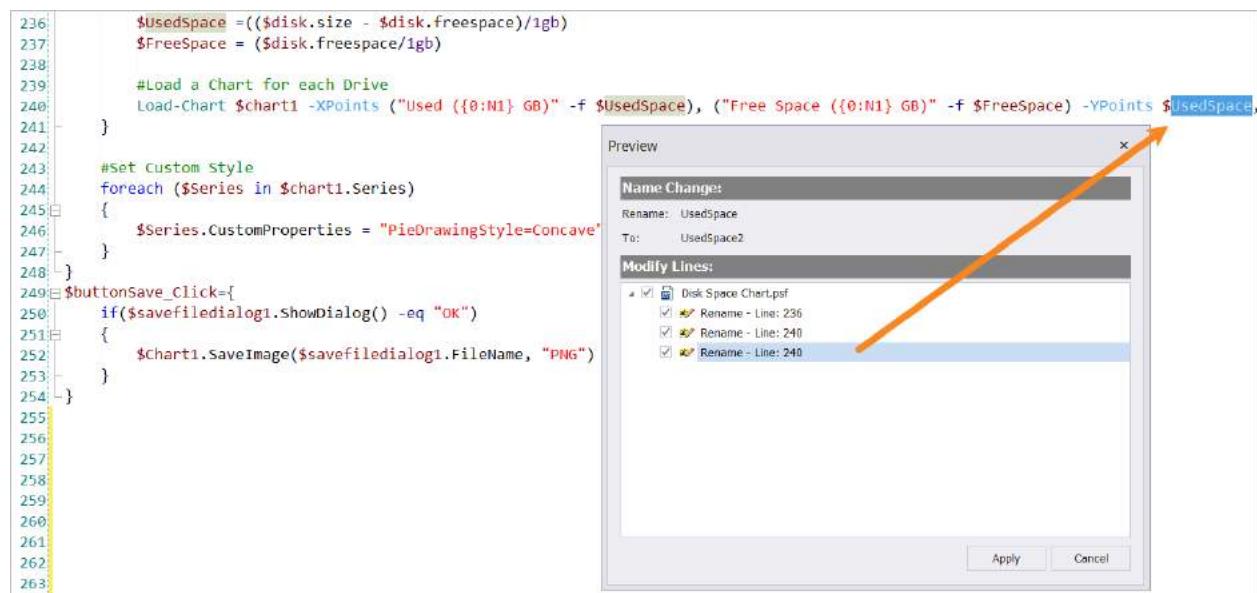
<b>Current Scope</b>	Rename refactoring occurs within the scope of the current function or event block.
<b>Entire Document</b>	Rename refactoring will occur for all instances of the object within the entire document.
<b>Entire Project</b>	Rename refactoring will occur for all instances of the object within the project files.

**i** GUI Controls, Events, Parameters, and Functions will always apply to the scope when *Entire Document* or *Entire Project* are selected.

- **Preview Reference Changes**

Presents the Preview dialog with a list of all the changes that will occur. This option is selected by default.

Click on any item listed in the Preview dialog to view the pending change in the script:



**!** When you select an item in the Preview dialog, notice that all of the other references in the script are also highlighted.

Uncheck an item if you do not want that particular instance to be modified. **i** If you are renaming a GUI control you will not be able to uncheck items.

To implement the selected changes, click **Apply** in the Preview dialog.

## Rename Refactoring in Projects

Rename refactoring is useful when dealing with multiple files in a project because it will update all of the project files, with the exception of excluded files (e.g., project files that have their Build property set to <*-ExcludeProperty*>):

The screenshot shows a code editor window with a script block. A context menu is open over the line '\$ChildForm\_comboboxDrives\_SelectedItem'. The 'Refactor' option is selected, which opens a 'Preview' dialog.

**Preview Dialog Content:**

- Name Change:**
  - Rename: ChildForm\_comboboxDrives\_SelectedItem
  - To: ChildForm\_comboboxDISKDrives\_SelectedItem
- Modify Lines:**
  - ChildForm.psf
    - Rename - Line: 12
    - Rename - Line: 21
    - Rename - Line: 30
  - MainForm.psf
    - Rename - Line: 11

**Code Editor Content (Lines 7-35):**

```
7 $buttonSelectRemovableDrive_Click={  
8     #TODO: Place custom script here  
9     if((call-ChildForm_psf -Filter "DriveType=2") -eq 'OK')  
10    {  
11        $ChildForm_comboboxDrives_SelectedItem  
12            Show-Drive  
13    }  
14 }  
15 fu  
16 {  
17 }  
18 }  
19 }  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35
```

In PowerShell Studio, return variables are updated when a GUI control is renamed. In addition, when you rename a project file in PowerShell Studio, all the reference functions and return variables are updated to reflect the new file name.

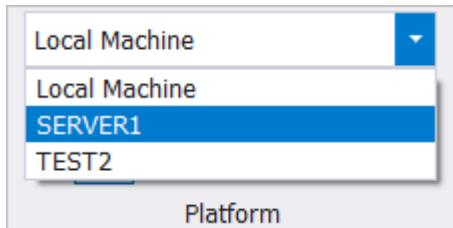
- i** If you include `ps1` files as content and dot source the file within a project, calls to functions defined in the dot sourced file can be updated using rename refactoring.

## 5.15 Verifying Your Script

PowerShell Studio provides a feature called Verify Script that checks if all of the cmdlets and modules required by your script are available on a target machine. This feature is especially useful for validating if your script will run on a specific remote machine.

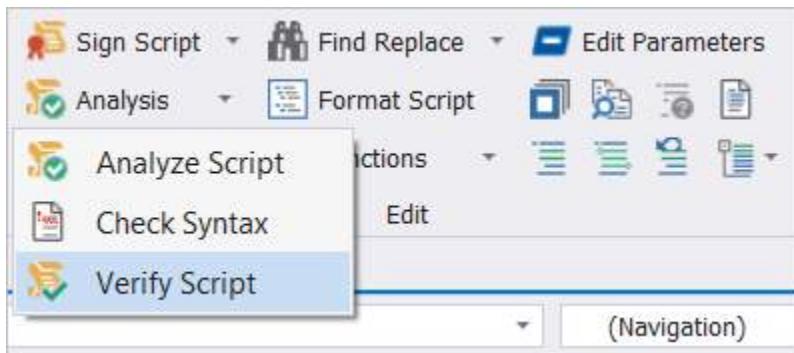
### To run Verify Script

Select a machine from the **Home** tab > **Platform** section, **Machine** drop-down:



- !** If you choose to select a remote machine you will need to import the remote machine's cache before selecting the remote machine (**Home** tab > **Platform** section > **Import Remote Cache** button).

Verify the script from the **Home** tab > **Edit** section > **Analysis** drop-down > **Verify Script**:



PowerShell Studio will evaluate the script and provide the results in the [Tools Output panel](#)<sup>254</sup>, listing any modules or cmdlets that are not on the target machine:

```
Tools Output
VerifyScriptTest.psf: -----
VerifyScriptTest.psf: Verify Script - VerifyScriptTest.psf
VerifyScriptTest.psf: -----
VerifyScriptTest.psf: Profile: REMOTE
VerifyScriptTest.psf:     PowerShell V5 (64 Bit)
VerifyScriptTest.psf: -----
VerifyScriptTest.psf: Missing Cmdlets:
VerifyScriptTest.psf: -----
VerifyScriptTest.psf:     Set-AppLockerPolicy
VerifyScriptTest.psf:     Get-AppLockerPolicy
VerifyScriptTest.psf:     Test-AppLockerPolicy
VerifyScriptTest.psf:     Import-Certificate
VerifyScriptTest.psf:     Get-Certificate
VerifyScriptTest.psf:     Export-Certificate
VerifyScriptTest.psf: -----
VerifyScriptTest.psf: Missing Modules:
VerifyScriptTest.psf: -----
VerifyScriptTest.psf:     AppLocker
VerifyScriptTest.psf:     PKI
```

PowerShell Studio's editor also offers a visual cue when a cmdlet is unknown:

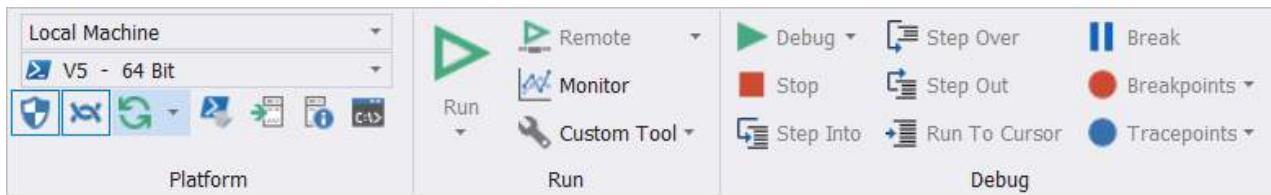
```
14 Import-Module PKI
15 Get-Certificate -LocalPath C:\Certificate.text
16 unknown Get-Certificate
17
```

## 6 Running and Debugging Scripts

This section covers some of the primary options available when running and debugging scripts in PowerShell Studio.

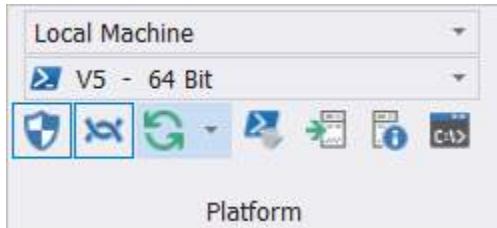
### 6.1 Run and Debug Ribbon Controls

Common controls related to running and debugging scripts are located on the Home tab of the ribbon. This section covers the options available in the [Platform](#)<sup>135</sup>, [Run](#)<sup>138</sup>, and [Debug](#)<sup>140</sup> sections of the Home ribbon.



#### Platform - Ribbon Options

The following options are available on the **Home** ribbon > **Platform** section:



*Home tab > Platform section*

- **Machine**

Select the machine to run the script on. If you import a remote cache, the remote machine name will be displayed in the drop-down list. 'Local Machine' is the default.

- [Powershell Version / Platform](#)<sup>136</sup>

Select the desired version of PowerShell, and 64 Bit or 32 Bit.

- [Enable / Disable Elevation](#)<sup>136</sup>

Toggles script execution and debugging in elevated mode.

- [STA Mode](#)<sup>137</sup>

Runs the script in Single Threaded Apartment (STA) mode.

- [Rebuild Local Cache](#)<sup>137</sup>

Rebuilds the local cache of PowerShell cmdlets and modules.

- [Reload Cache](#) 138

Reloads the PowerShell cache.

- [Cache Editor](#)

Add / remove modules that are included in the local cache profile.

- [Import Remote Cache](#)

Imports the Installed Module Set (IMS) exported on another computer.

- [Edit Remote Connection](#)

Edit the Remote Cache's Connection Settings.

- [Remote Console](#)

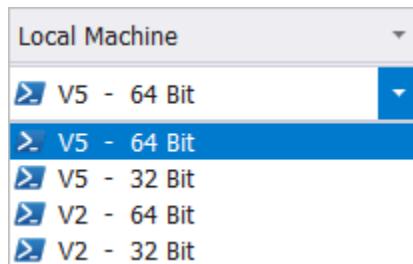
Open a remote shell to the selected machine (requires Windows Remoting).

## Powershell Version and Platform

---

PowerShell comes in both 64-bit and 32-bit platforms. When you execute a script from PowerShell Studio, you can choose the required platform from the **Home** tab > **Platform** section.

You can also select the version of PowerShell to run the script under:



You can also designate how and where a script will run [using meta comments](#) 141

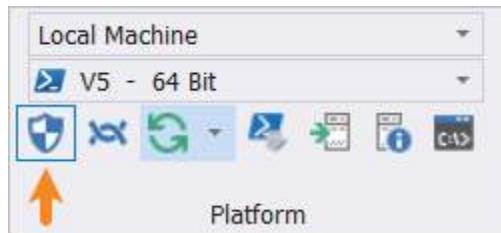
## Elevation

---

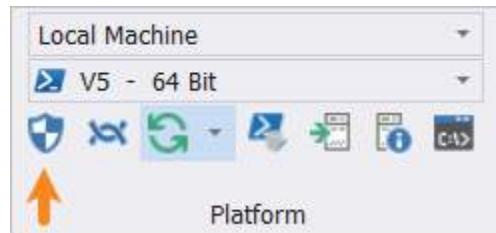
PowerShell Studio runs your scripts with the privileges of the current user. It is considered a security best practice to avoid logging on as an 'administrator' level account whenever possible. Sometimes your scripts will need to do things that require greater privileges—PowerShell Studio facilitates this by allowing you to run scripts in *elevated* mode.

You can toggle between the elevated or non-elevated mode for script execution from the **Home** tab > **Platform** section > **Enable/Disable Elevation** button:

## Elevated



## Not Elevated



## STA Mode

STA (Single Threaded Apartment) mode allows you to start your script in single threaded mode. This is essential when your script uses forms to interact with the Windows GUI. Some GUI controls require STA mode in order for them to function correctly. STA mode is activate by default.

You can toggle the STA Mode between active and inactive from the **Home** tab > **Platform** section > **STA Mode** button:

### STA Mode - Active



### STA Mode - Inactive

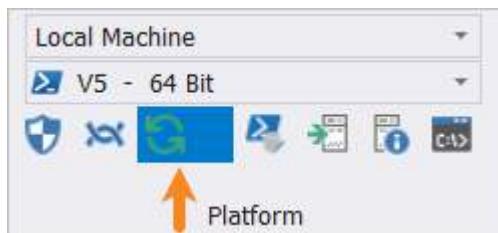


## Cache Commands

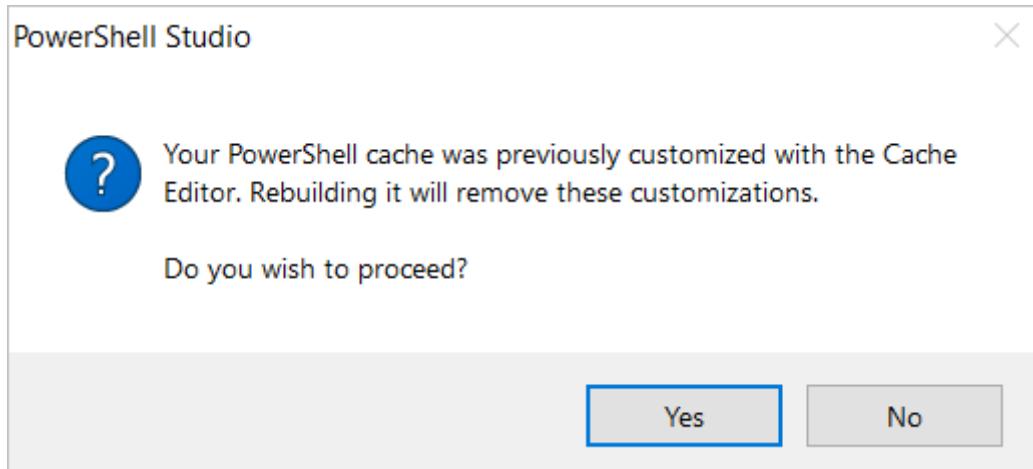
There are two cache related commands:

- **Rebuild Local Cache**

This command rebuilds the cache, including any new modules installed:



If you modified the cache with Cache Editor, PowerShell Studio will now prompt you before re-building the cache:

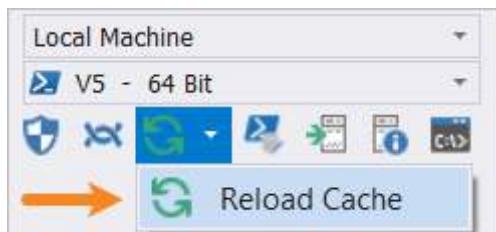


- To override the manual changes, select **Yes**.
- To cancel, Select **No**.

👉 This safety check will help prevent accidentally undoing any manual changes made to the cache.

- **Reload Cache**

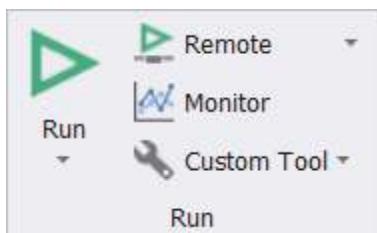
This command will reload the cache without making any changes:



👉 Use this command to reload the cache after making changes with the Cache Editor.

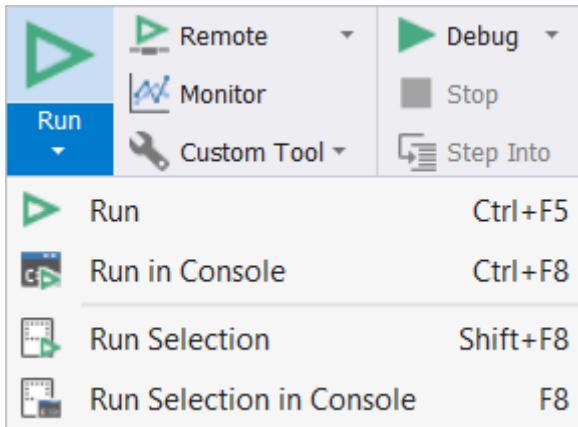
## Run - Ribbon Options

The following script execution options are available on the **Home** ribbon > **Run** section:



*Home tab > Run section*

- **Run**



- **Run (Ctrl+F5)**

Executes the current document or project. The results are displayed in the Output panel.

- **Run in Console (Ctrl+F8)**

Executes the script or project in a console session. The results are displayed in the Console panel.

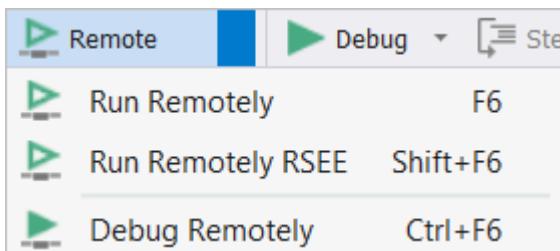
- **Run Selection (Shift+F8)**

Executes the highlighted text or the line that it is on. The results are displayed in the Output panel.

- **Run Selection in Console (F8)**

Executes the highlighted text in a console session. The results are displayed in the Console panel.

- **Remote**



- **Run Remotely (F6)**

Uses PowerShell Remoting to execute the script or project on another machine. The results are displayed in the Output panel.

- **Run Remotely RSEE (Shift+F6)**

Uses the SAPIEN *Remote Script Execution Engine* to execute the file on another machine.

- **Debug Remotely (Ctrl+F6)**

Debugs the current script or project on a remote system. The results are displayed in the Output panel and the Debug Console.

- **Monitor**

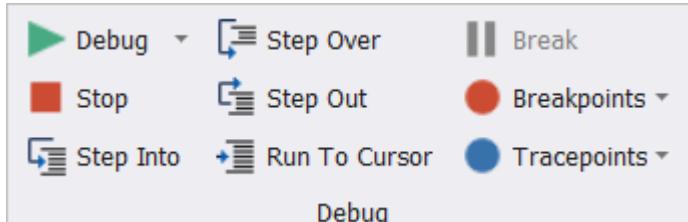
Enables performance monitoring when running scripts and displays the output in the [Performance panel](#).

- **Custom Tool**

User defined menu of commands and tools.

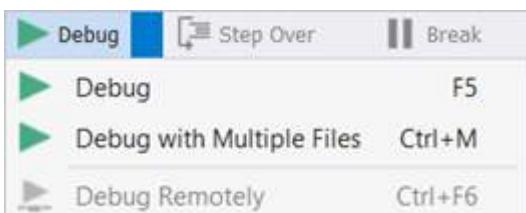
## Debug - Ribbon Options

The following debug options are available on the **Home tab > Debug** section:



*Home tab > Debug section*

- **Debug**



- **Debug (F5)**

Debugs the current script or project. The results are displayed in the Output and Tools Output panels.

- **Debug with Multiple Files (Ctrl+M)**

Debugs the current script or project plus additional files and their breakpoints.

- **Debug Remotely (Ctrl+F6)**

Debugs the current script or project on a remote system. The results are displayed in the Output panel and the Debug console.

- **Stop (Shift+F5)**

Stops the running script.

- **Step Into (F11)**

Step into the current function call.

- **Step Over (F10)**

Single step.

- **Step Out (Shift+F11)**

Steps out of the current function.

- **Run To Cursor (Ctrl+F10)**

Runs the script to the line containing the cursor.

- **Break**  
Breaks into the debugger.
- **Breakpoints**  
See [Working with Breakpoints](#)<sup>[142]</sup>.
- **Tracepoints**  
See [Working with Tracepoints](#)<sup>[144]</sup>.

## 6.2 Running Scripts

Many of the controls used when running scripts are located in the [Platform](#)<sup>[135]</sup> and [Run](#)<sup>[138]</sup> sections of the Home ribbon. These controls are also [contextually available](#)<sup>[83]</sup> when you right-click in a script.

### Using Meta Comments

PowerShell Studio allows meta comments to be used to change how and where your scripts are run. You can run scripts in 32 or 64-bit mode, elevated or not elevated, remote or local, and so on. These meta comments override your current platform settings. Many options can be changed or set via meta comments such as:

- **# %ForceShell% = PowerShell 64 bit**  
Will invoke the PowerShell 64-bit shell and execute the script there.
- **# %ForcePlatform% = 64**  
Will execute your script in 64-bit mode.
- **# %ForceElevation% = yes**  
Will force elevation of your script.
- **# %ForceHost% = REMOTE1**  
Will run the script on a remote machine named 'REMOTE1'.

Meta comments can be combined. For example, the following will cause the script to be run in 64-bit mode and elevated:

```
6
7 # %ForcePlatform% = 64
8 # %ForceElevation% = yes
9
10 function Get-Scriptdirectory
```

You can also specify optional credentials. For instance, the following meta comments will force the script to run on the remote machine 'REMOTE1' and will prompt for user name and password:

```
# %ForceHost% = REMOTE1, Prompt
```

The following meta comments will force the script to run on 'REMOTE1', but will only prompt for the password for the user named 'User1':

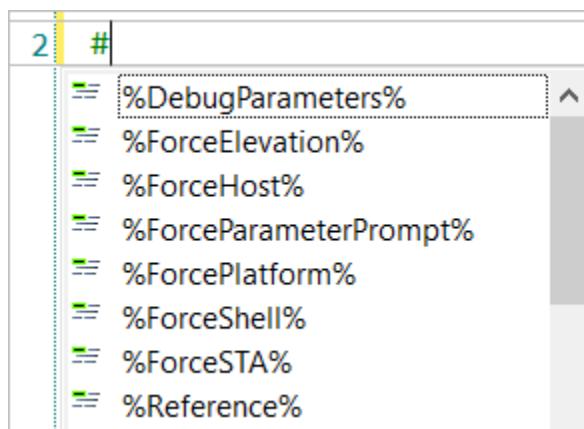
```
# %ForceHost% = REMOTE1, User1, Prompt
```

## Available Meta Comments

- %DebugParameters%
- %ForceElevation% = true | false
- %ForceHost% = Hostname[,User[,password]]
- %ForceParameterPrompt% = true | false
- %ForcePlatform% = 32 | 64
- %ForceShell% = Name
- %ForceSTA% = true | false
- %Reference%

 The %Reference% meta comment only affects PrimalSense—it will load the specified assembly and provide PrimalSense for the .NET types contained within.

 PrimalSense will offer suggestions when you type # in the code editor:



## 6.3 Debugging Scripts

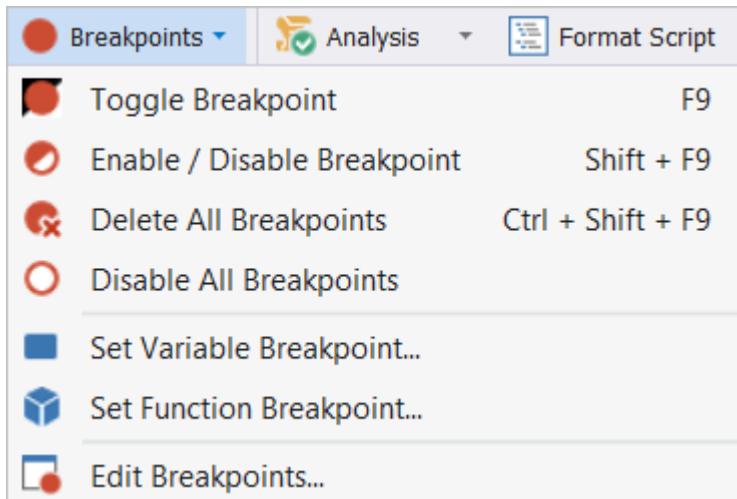
This section provides an overview of tasks performed during debugging.

### 6.3.1 Working with Breakpoints

Breakpoints instruct the debugger to stop on a specified line of code, allowing you time to review what the script is doing at that point.

## Breakpoints - Menu Options

The Breakpoints drop-down menu on the Home ribbon provides the following options:



- **Toggle Breakpoint (F9)**  
Toggles the breakpoint on the current line.
- **Enable / Disable Breakpoint (Shift+F9)**  
Enables or disables the breakpoint on the current line.
- **Delete All Breakpoints (Ctrl+Shift+F9)**  
Deletes all breakpoints in the active document.
- **Disable All Breakpoints**  
Disables all breakpoints in the active document.
- **Set Variable Breakpoint...**  
Sets an advanced breakpoint when a variable is modified or accessed.
- **Set Function Breakpoint...**  
Sets an advanced breakpoint when a specific function or command is called.
- **Edit Breakpoints...**  
Opens a dialog to view and remove breakpoints.

### To set a breakpoint

- **Left-click** in the grey margin to the left of a line number.  
**-OR-**
- Place your cursor on a line and select any of these options:
  - Press **F9**.
  - Right-click to access the context menu > select **Breakpoints** > select **Toggle Breakpoint**.

- On the Home ribbon > in the **Debug** section > click the **Breakpoints** drop-down menu > select **Toggle Breakpoint**.

Active breakpoints are displayed as solid red circles in the margin of the code editor window, and disabled breakpoints are displayed as red rings:

```
235 $datagridviewResults_ColumnHeaderMouseClick=[System.Windows.Forms.DataGridViewColumnHeaderMouseClickEventArgs]
236 #Event Argument: $_ = [System.Windows.Forms.DataGridViewCellEventArgs]
237 if($datagridviewResults.DataSource -is [System.Data.DataTable])
238 {
239     $column = $datagridviewResults.Columns[$_.ColumnIndex]
240     $direction = [System.ComponentModel.ListSortDirection]::Ascending
241
242     if($column.HeaderCell.SortGlyphDirection -eq 'Descending')
243     {
244         $direction = [System.ComponentModel.ListSortDirection]::Descending
```

## To disable or delete a breakpoint

- Left-click in the grey margin to the left of a line number containing a breakpoint.

-OR-

- Place your cursor on a line containing a breakpoint and select any of these options:
  - Press **F9** (**Shift+F9**).
  - Right-click to access the context menu > select **Breakpoints** > select **Toggle Breakpoint** or **Enable / Disable Breakpoint**.
  - On the Home ribbon > in the **Debug** section > click the **Breakpoints** drop-down menu > select **Toggle Breakpoint** or **Enable / Disable Breakpoint**.

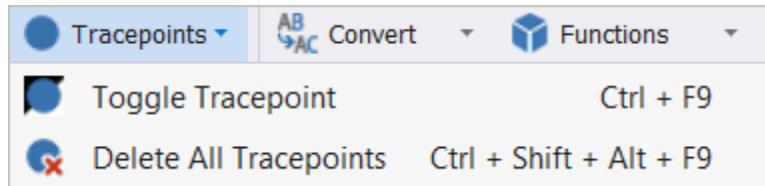
Use the **F9** or left-click toggle functions mentioned above to cycle through the **Enable**, **Disable**, and **Delete** breakpoint options.

## 6.3.2 Working with Tracepoints

Tracepoints cause PowerShell Studio to write a message to the [Output panel](#)<sup>221</sup> when a particular line of code is executed.

### Tracepoints - Menu Options

The Tracepoints drop-down menu on the Home ribbon provides the following options:



- **Toggle Tracepoint (Ctrl+F9)**  
Toggles the tracepoint on the current line.
- **Delete All Tracepoints (Ctrl+Shift+Alt+F9)**  
Removes all tracepoints in the active document.

## To set a tracepoint

---

- Place your cursor on a line and select any of these options:
  - Press **Ctrl+F9**.
  - Right-click to access the context menu > select **Tracepoints** > select **Toggle Tracepoint**.
  - On the **Home** ribbon > in the **Debug** section > click the **Tracepoints** drop-down menu > select **Toggle Tracepoint**.

Tracepoints appear as solid blue circles in the code editor margin:

```
14 Filter ReverseString {  
15     $CharArray = $_.ToCharArray()  
16     For ($i = $CharArray.Count; $i -ge 0; $i--) {  
17         Write-Output $CharArray[$i] -NoNewLine  
18     }  
19     Write-Output ""  
20 }  
21  
22 "SAPIEN" | ReverseString
```

The screenshot shows a PowerShell script in a code editor. A solid blue circle, representing a tracepoint, is positioned in the margin next to the opening brace of the 'ReverseString' function on line 14. The code itself is a simple filter that takes an input string and outputs it in reverse order, with a final newline character.

When the code is executed, the tracepoint output is displayed in the Output panel:

```
Output
>> Debugging (Filter_Test-1.ps1) Script...
>> Platform: V5 64Bit (STA)
Filter_Test-1.ps1 (17): Tracepoint: Line 17 at 4:14:33 PM
-NoNewLine
Filter_Test-1.ps1 (17): Tracepoint: Line 17 at 4:14:33 PM
N
-NoNewLine
Filter_Test-1.ps1 (17): Tracepoint: Line 17 at 4:14:33 PM
E
-NoNewLine
Filter_Test-1.ps1 (17): Tracepoint: Line 17 at 4:14:33 PM
I
-NoNewLine
Filter_Test-1.ps1 (17): Tracepoint: Line 17 at 4:14:33 PM
P
-NoNewLine
Filter_Test-1.ps1 (17): Tracepoint: Line 17 at 4:14:33 PM
A
-NoNewLine
Filter_Test-1.ps1 (17): Tracepoint: Line 17 at 4:14:33 PM
S
-NoNewLine

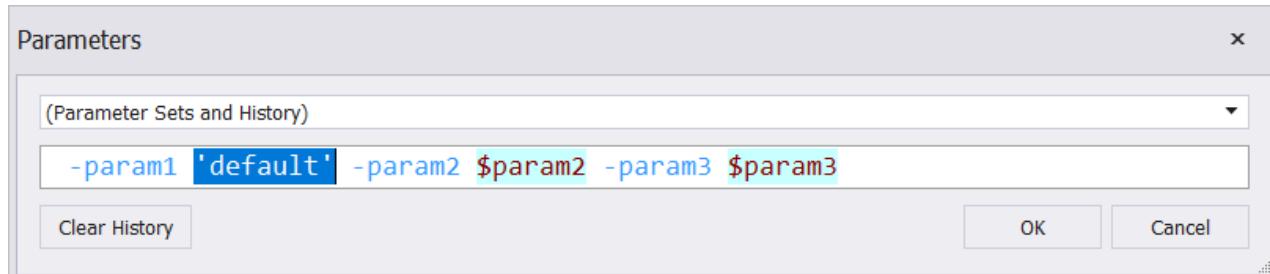
>> Script Ended
```

## To delete a tracepoint

- Place your cursor on a line containing a tracepoint and select any of these options:
  - Press **Ctrl+F9**.
  - **Right-click** to access the context menu > select **Tracepoints** > select **Toggle Tracepoint**.
  - On the **Home** ribbon > in the **Debug** section > click the **Tracepoints** drop-down menu > select **Toggle Tracepoint**.

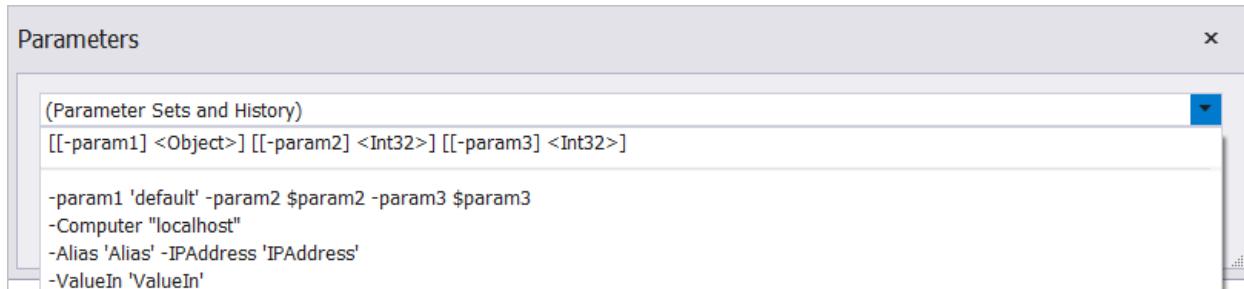
### 6.3.3 Passing Parameters

If your script begins with a *Param* block, PowerShell Studio will display a dialog box to allow you to select or enter values for script parameters:



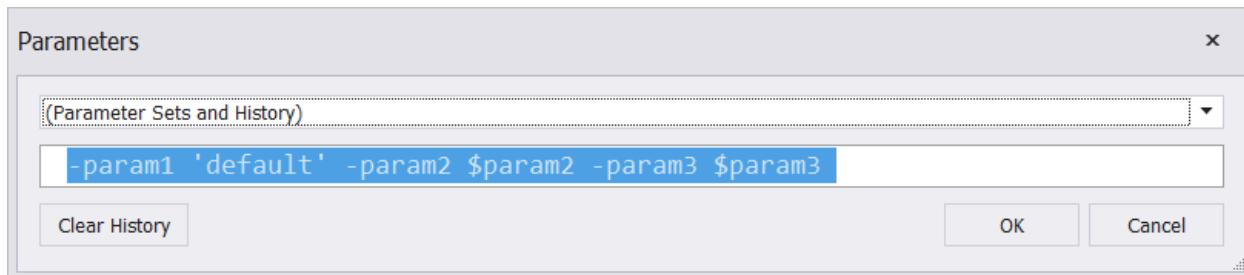
#### To select or enter values in the Parameters dialog

- Select previously stored parameter values from the *Parameter Sets and History* drop-down:



-OR-

- Enter a new set of parameter values in the second text box with a space between each value:



Click **OK** to pass the selected values to the script.

- The **Clear History** button allows you to remove all of the previously stored parameter values.

## 6.3.4 Debug Panels

During a debugging session, PowerShell Studio displays a collection of panels to help you resolve problems in your code.

### Panels used during debugging

#### [Call Stack](#)

Displays the function or procedure calls that are currently on the stack.

#### [Debug Console](#)

Customizable command line console (PowerShell, PSCore, Bash, etc.) that allows you to interact with a debug session when at a breakpoint.

#### [Output](#)

Displays all script output including general application messages, build information, errors, debug, verbose, and tracepoint output.

#### [Tools Output](#)

Displays output from external tools. When debugging, displays breakpoint notifications and post mortem messages.

#### [Variables](#)

Lists all variables and values in the current scope during a breakpoint when debugging.

#### [Watch](#)

Displays the values of variables and expressions that you define when debugging.

## 6.4 Running and Debugging Remotely

PowerShell Studio provides a number of options for running scripts or packages on remote machines, and also allows you to debug remotely.

### Running Scripts Remotely

PowerShell Studio supports two different mechanisms for executing scripts or packages on a remote machine:

- [PowerShell Remoting](#)
- [RSEE \(Remote Scripting Execution Engine\) Remoting](#)

### Using PowerShell Remoting

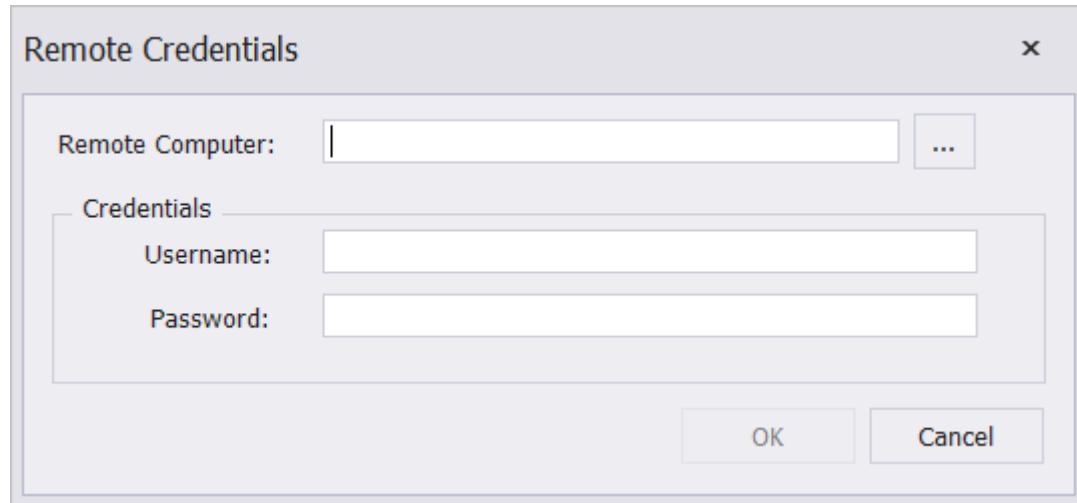
PowerShell Remoting must be configured on the target machine for remote script execution.

*For more information, [view a simple guide to installing and configuring PowerShell Remoting](#).*

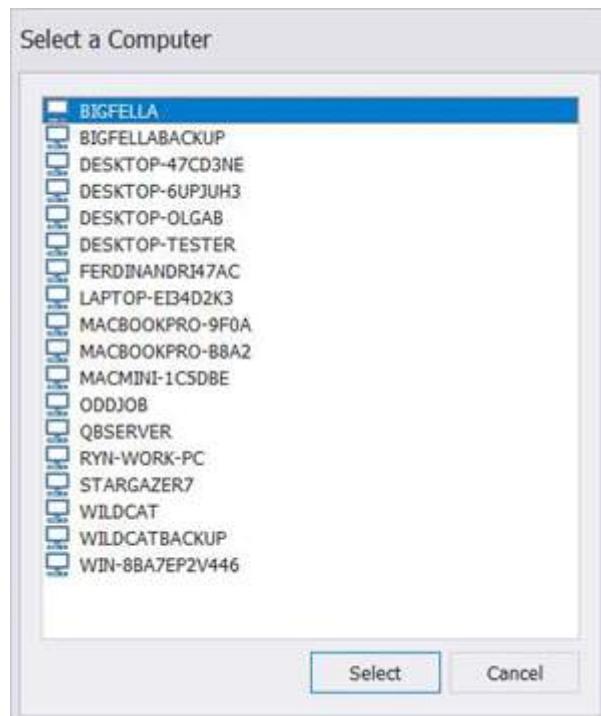
## To execute a file using PowerShell Remoting

- On the **Home** tab > in the **Run** section, click the **Remote** drop-down, then select **Run Remotely** (**F6**).

The Remote Credentials dialog will open:



- Either enter the name of the remote computer, or press the browse button to launch the Select a Computer window:



- After selecting a computer, enter the Username and Password credentials and then select OK. The script will be executed on the selected machine and the results will be displayed in the **Output** panel.
- i** When you run a script remotely it cannot interact with the desktop, therefore you cannot include any code that prompts the user for input (such as Read-Host, Get-Credential, or any forms).

## Using RSEE Remoting

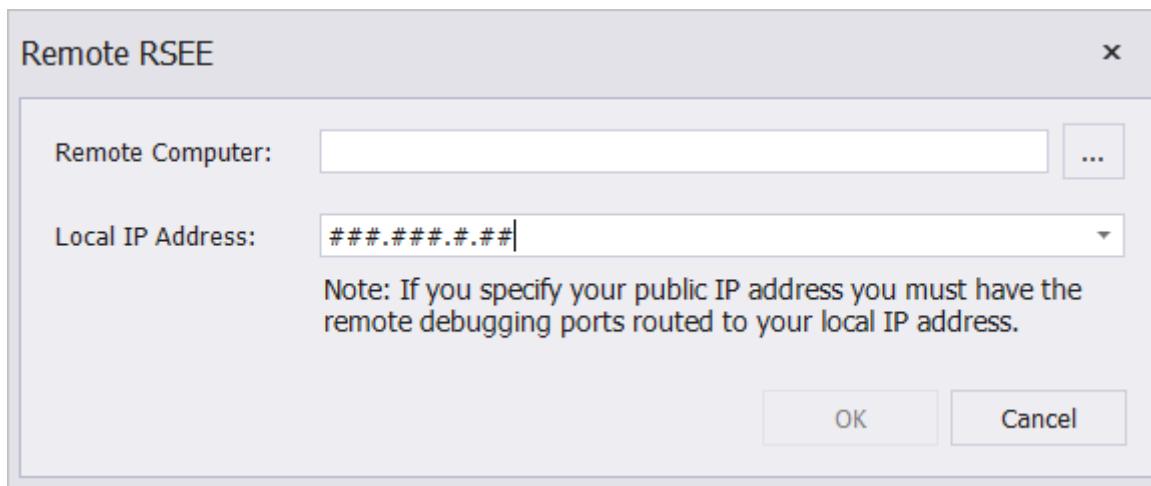
SAPIEN Technologies' [RSEE service](#)<sup>376</sup> must be installed and running on the target machine to use RSEE remoting. The installation files (both 32 and 64-bit) can be found in:

`%Program Files%\SAPIEN Technologies, Inc\PowerShell Studio <year>\Redistributables`

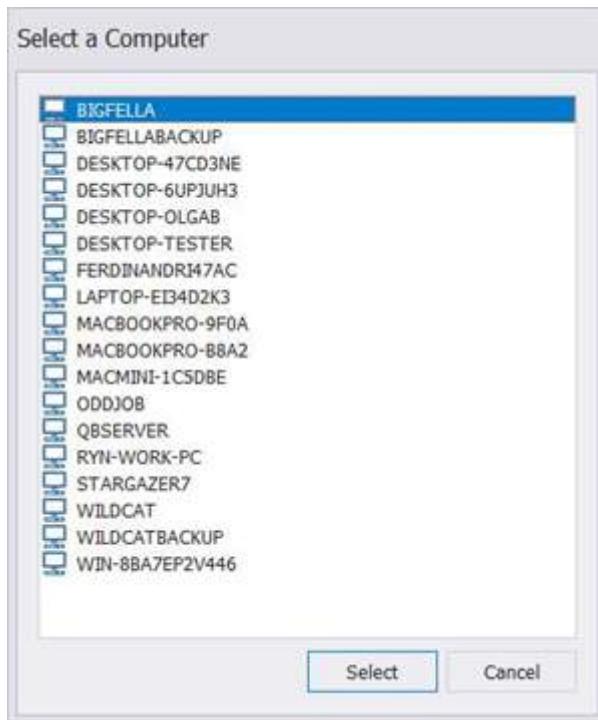
### To execute a file using RSEE Remoting

- On the **Home** tab > in the **Run** section, click the **Remote** drop-down, then select **Run Remotely RSEE (Shift+F6)**.

The Remote RSEE dialog will open:



- Either enter the name of the remote computer, or press the browse button to launch the Select a Computer window:



- Choose a remote computer, then click **Select**. The script will be executed on the selected machine and the results will display in the **Output** panel.

## Remote Debugging

PowerShell Studio allows you to debug scripts as they run on another machine using SAPIEN Technologie's *Remote Scripting Execution Engine* (RSEE). RSEE must be installed on any machine that will host remote scripts. The installation files (both 32 and 64-bit) can be found in:

`%Program Files%\SAPIEN Technologies, Inc\PowerShell Studio <year>\Redistributables`

## 7 GUI Designer

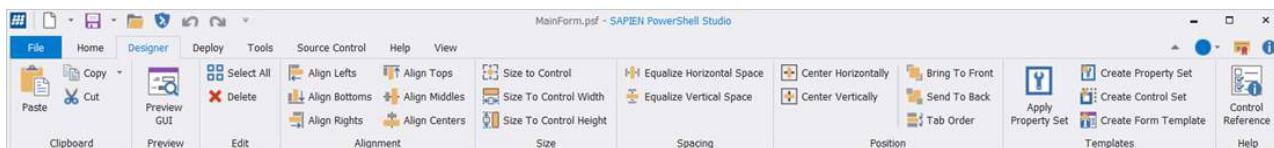
PowerShell Studio fully supports the creation of GUI scripts based on Windows Forms technology. A number of predefined forms are included to get you started, or you can start with a blank form and build everything from scratch.

### 7.1 Forms Designer Introduction

The main editor screen for GUI scripts is the Forms Designer. This topic introduces you to the Designer ribbon, and also the panels associated with the Designer.

#### Designer Ribbon

The Designer tab consolidates common tasks performed when designing and editing GUI forms:



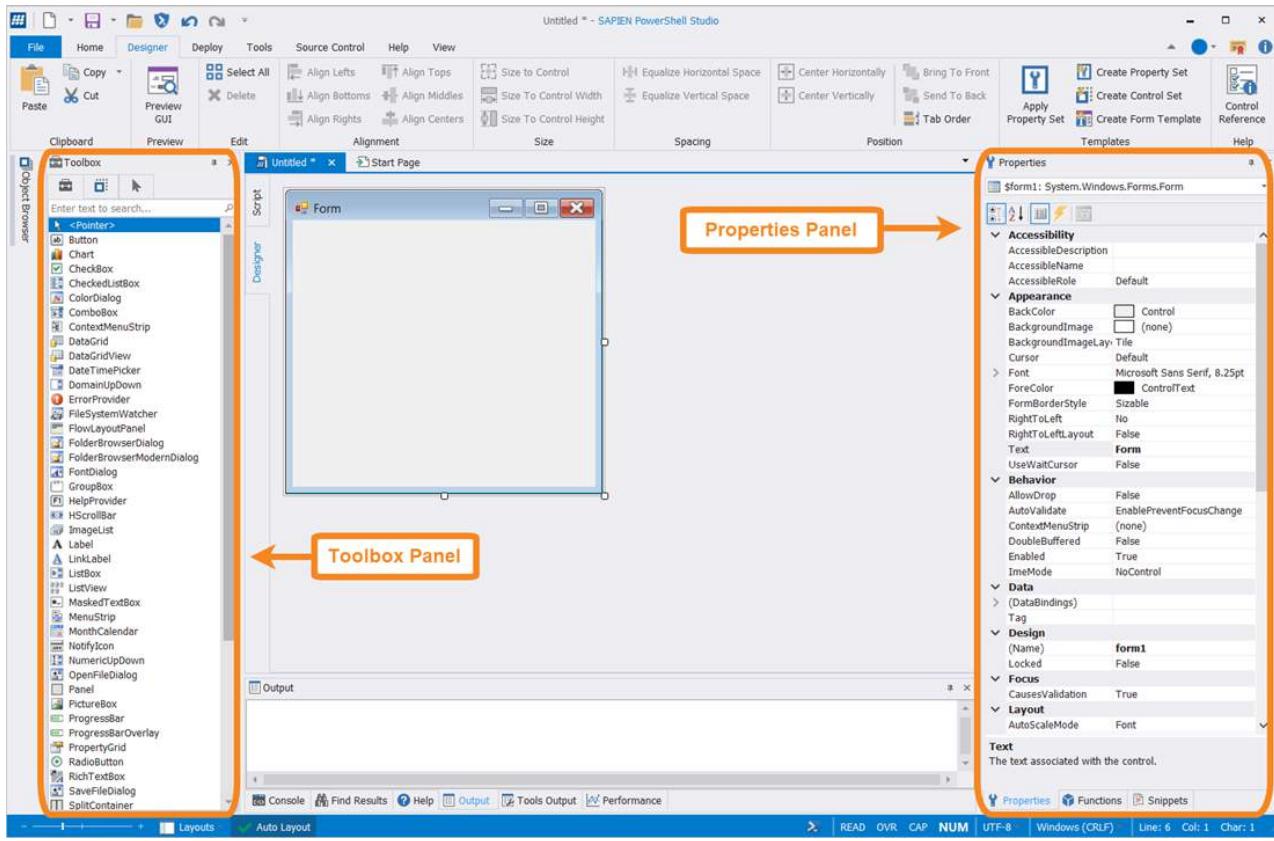
**i** The tasks available on the Designer ribbon correlate to what is selected in the form. For example, selecting one form control will activate all of the tasks in the Position section, and selecting more than one form control will active all of the tasks in the Alignment, Size, and Spacing sections.

The controls available on the Designer ribbon are covered in the relevant topics throughout the [GUI Designer](#)<sup>152</sup> section.

#### Designer Panels

The [Toolbox](#)<sup>250</sup> and [Properties](#)<sup>240</sup> panels are used when working in the Forms Designer:

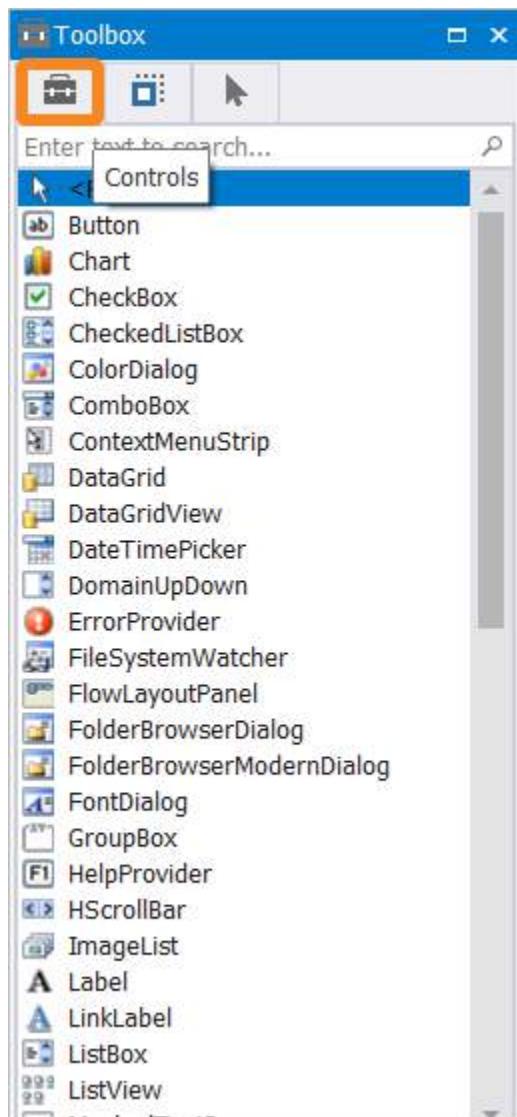
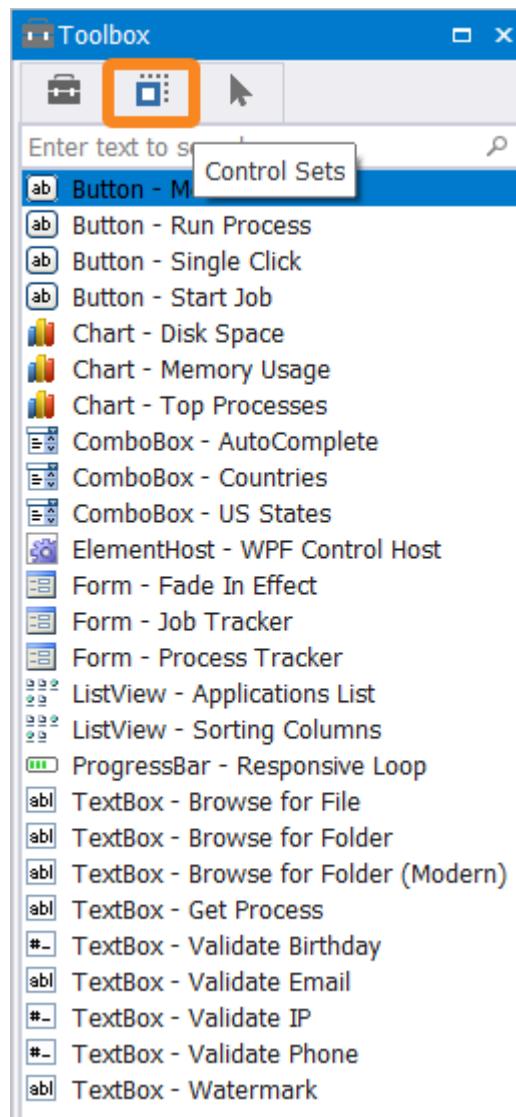
# GUI Designer



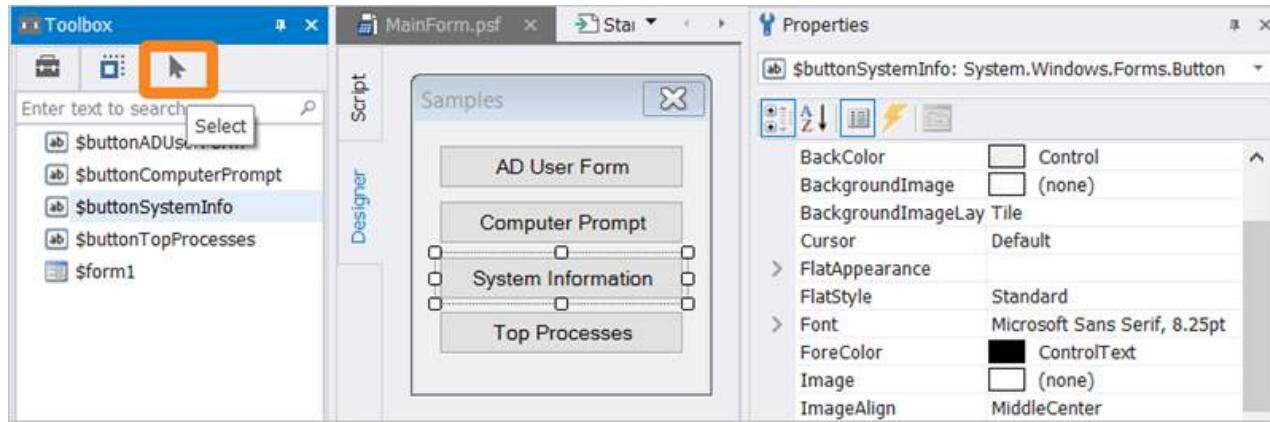
## Toolbox Panel

The [Toolbox panel](#) [250] provides lists of controls and control sets that can be used when designing a PowerShell form.

- Controls are built in .NET controls.
- Control sets are custom controls built out of standard controls and custom scripts.

**Controls****Control Sets****Select Tool**

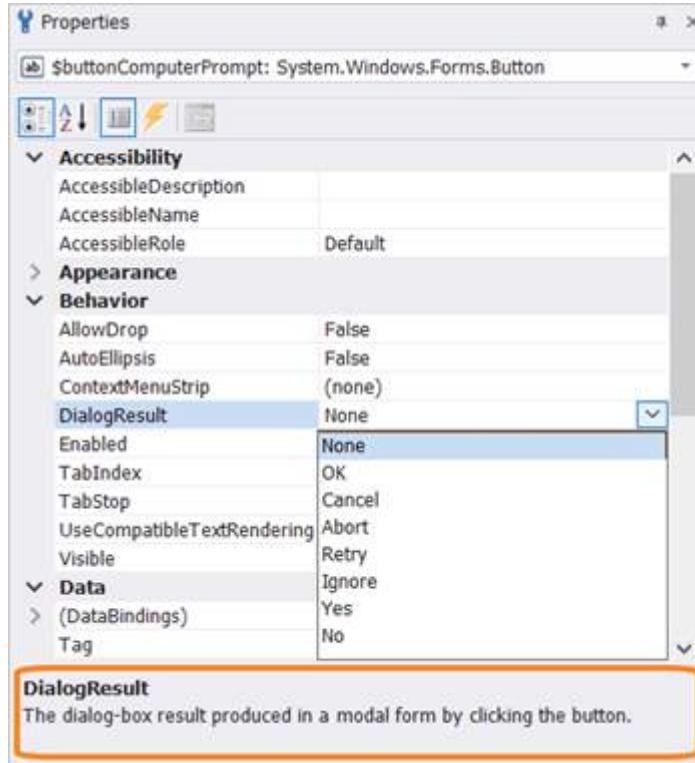
The *Select* tab is on the top of the Toolbox panel, to the right of the Control Sets tab. Use the Select tool to select a control, which selects the control in the Designer and displays the control properties:



Use Shift+Click or Ctrl+Click to select multiple controls. When multiple controls are selected, the Properties panel displays only the properties that the selected controls have in common.

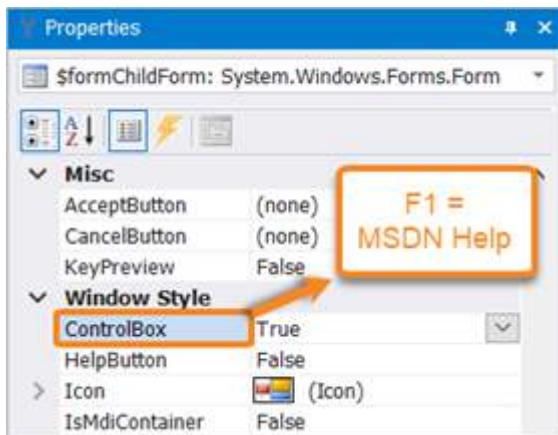
### Properties Panel

The [Properties panel](#)<sup>[240]</sup> allows you to view and edit the properties of the currently selected control. Each property has an associated editor to help guide you in choosing the correct value:

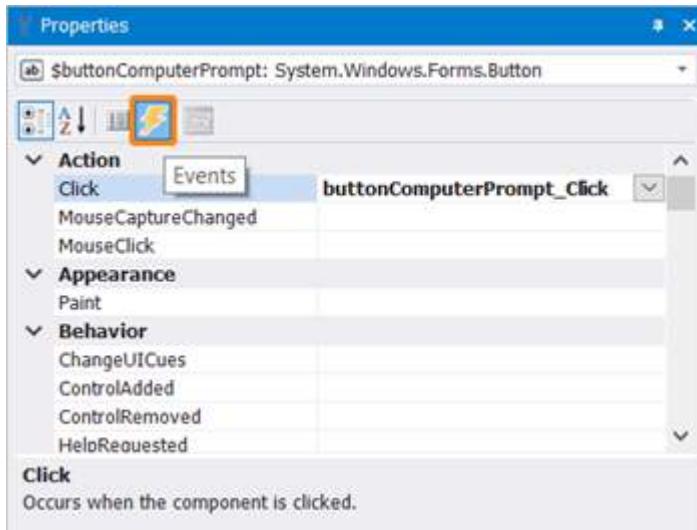


When you edit form controls in the Properties panel the changes are immediately reflected in the Designer.

👉 Pressing F1 when a control property is selected will open the related MSDN Help topic in a browser window:



The Properties panel also displays the events that a control can respond to, and allows you to connect an event to code:

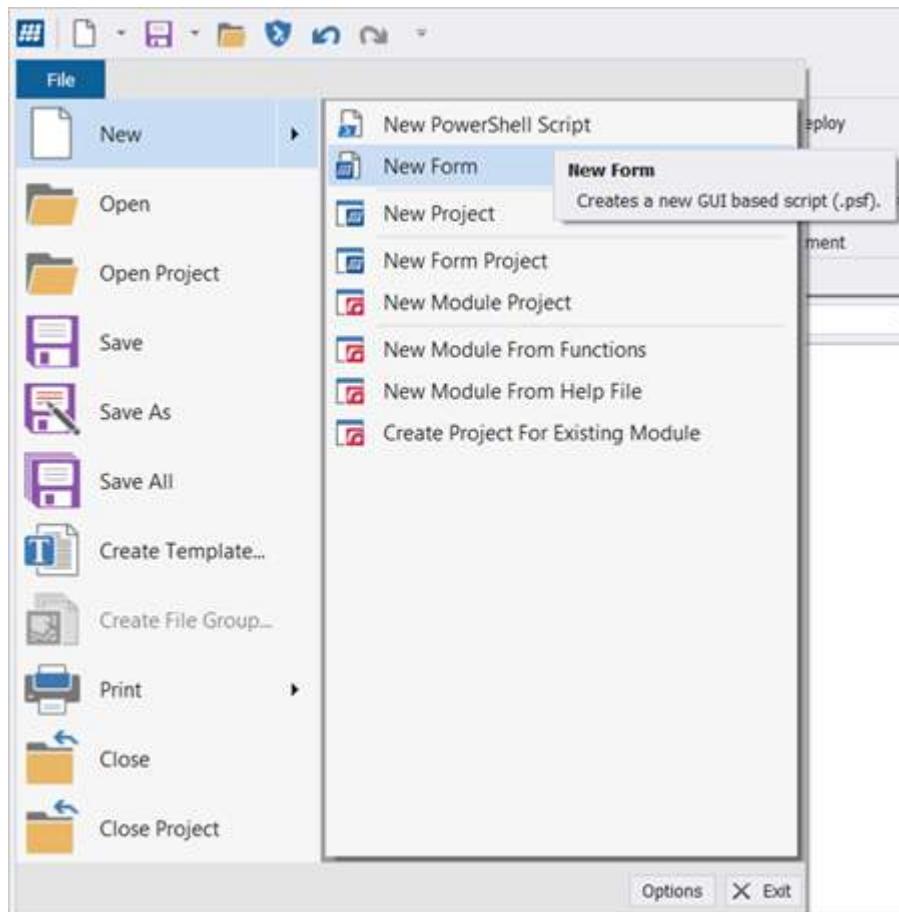


## 7.2 Creating a New Form

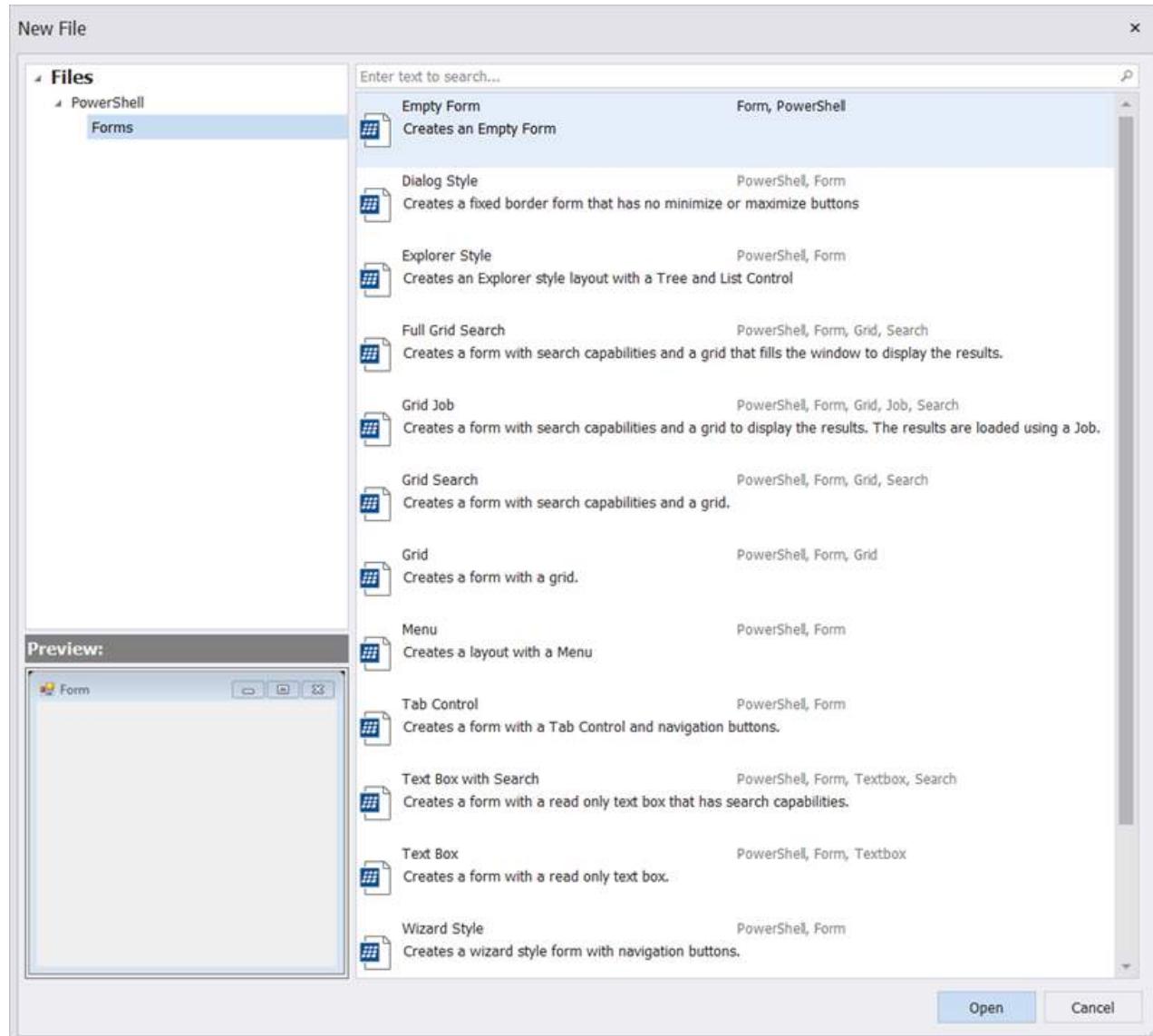
When creating a new form, you can start with an empty form or choose from a [predefined template](#) .

### To design a new form

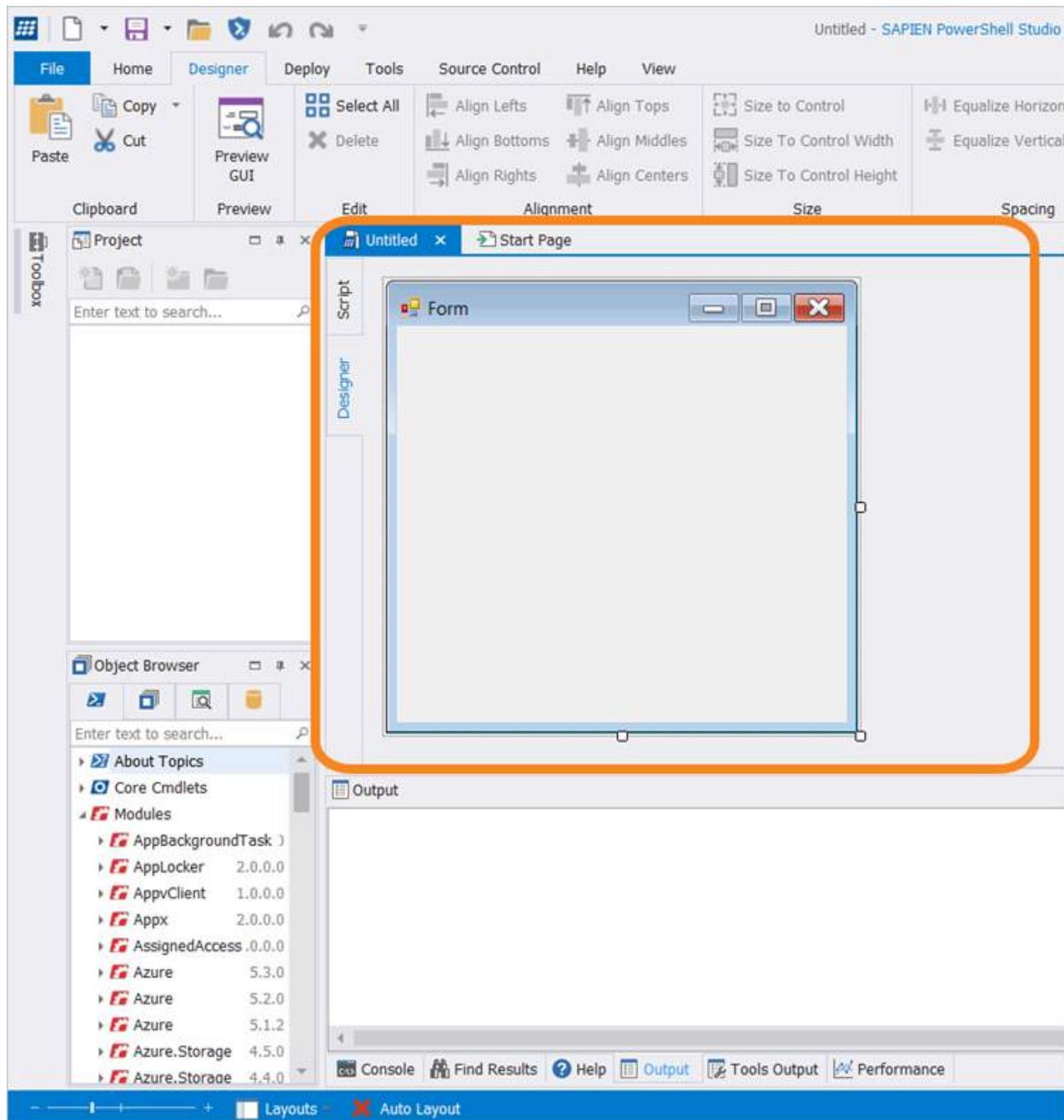
Select File > New > New Form:



Select a form in the New File dialog, and then click **Open**:



The form will open in the Designer window of the main editor:



## 7.3 Working with Form Controls

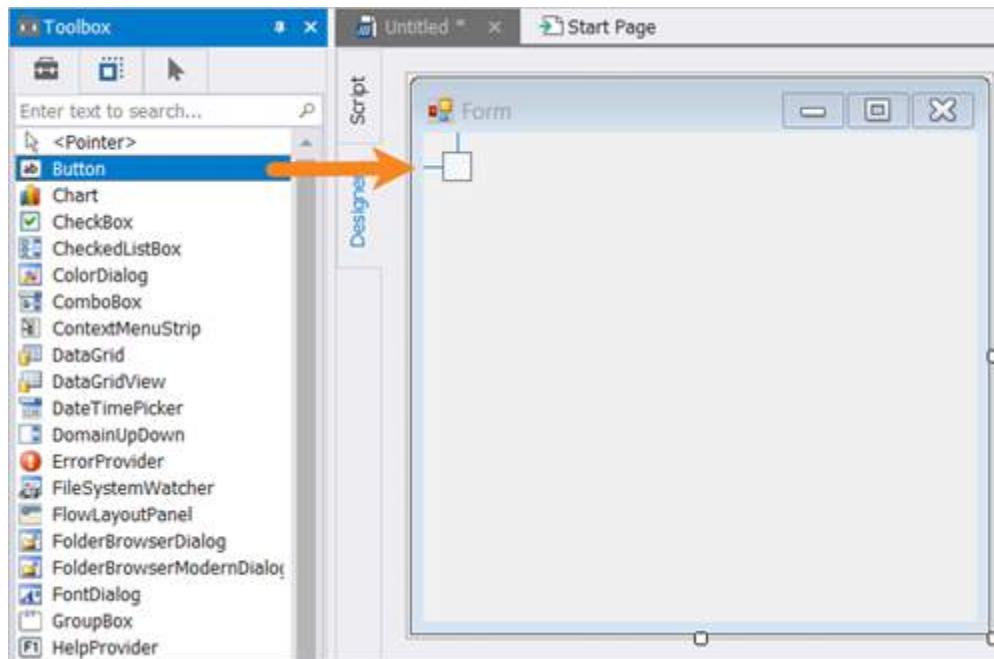
This section shows you how to add a control or control set to a form, and how to work with form controls.

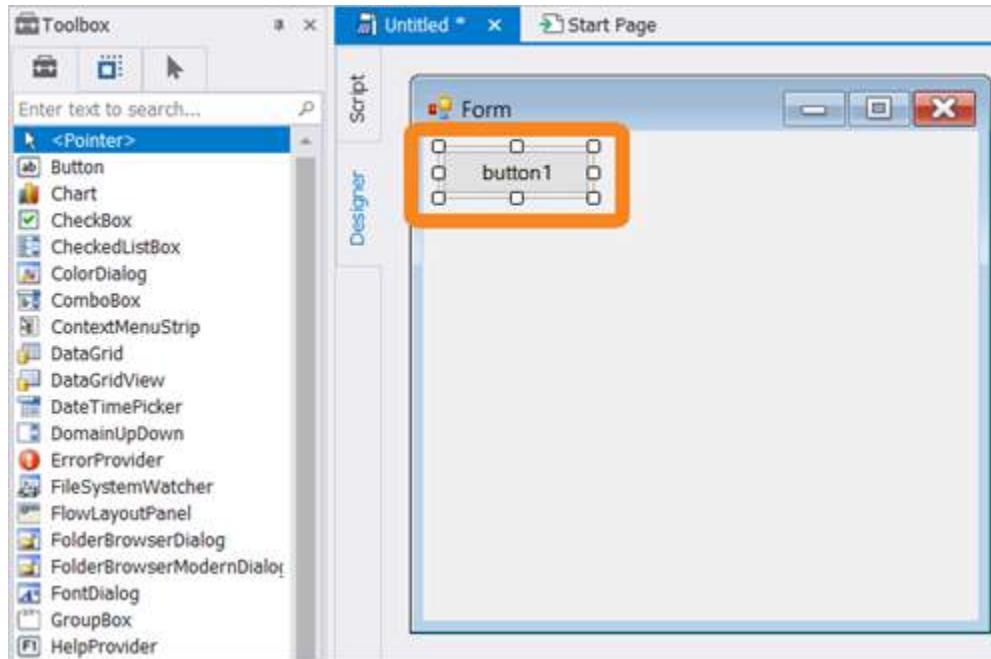
## Adding Form Controls

### To add a control or control set to a form

- Double-click the control or control set.
- Drag and drop the control or control set onto the form.
- Right-click on the control or control set and select **Insert**.

These images show a button control being dragged and dropped onto an empty form:





## Manipulating Controls

The Edit section of the Designer ribbon contains a number of useful commands for working with controls.



From left to right:

- **Edit** - Select or remove controls:
  - **Select All** - Selects all controls on a form (see [Working with Multiple Controls](#) below).
  - **Delete** - Delete(s) the selected controls.
- **Alignment** - Align controls to a particular edge:
  - **Align Lefts** - Align left edges.
  - **Align Bottoms** - Align bottom edges.
  - **Align Rights** - Align right edges.
  - **Align Tops** - Align top edges.
  - **Align Middles** - Align middles.
  - **Align Centers** - Align centers.

- **Size** - Resize a control to its content:
  -  **Size to Control** - Size to control.
  -  **Size to Control Width** - Size to control width.
  -  **Size to Control Height** - Size to control height.
- **Spacing** - Equalize spacing between controls:
  -  **Equalize Horizontal Space** - Makes the horizontal distances between the selected controls equal.
  -  **Equalize Vertical Space** - Makes the vertical distances between the selected controls equal.
- **Position** - Location of the controls on a form:
  -  **Center Horizontally** - Center controls horizontally on a form.
  -  **Center Vertically** - Center controls vertically on a form.
  -  **Bring to Front** - Move controls in front of other controls.
  -  **Send to Back** - Move controls behind other controls.
  -  **Tab Order** - Set the order in which a user moves focus from one control to another by pressing the Tab key.

## Working with Multiple Controls

Working with multiple controls involves selecting more than one form control and applying the same change to all selected controls.

There are a number of ways to select multiple controls at once:

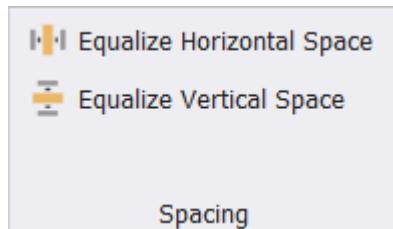
- Click on the first control, and then **Ctrl+Left-click** or **Shift+Left-click** to select the other controls.
- Left-click and drag to "lasso" and select individual controls as a group.
- On the Designer ribbon in the Edit section, click **Select All**  to select all of the controls on a form.

Once selected, any changes you make in the Properties panel or the Add Events dialog will be applied to all of the controls.

-  If you select more than one type of control, the Properties panel will only display the properties and events that all of the controls share.

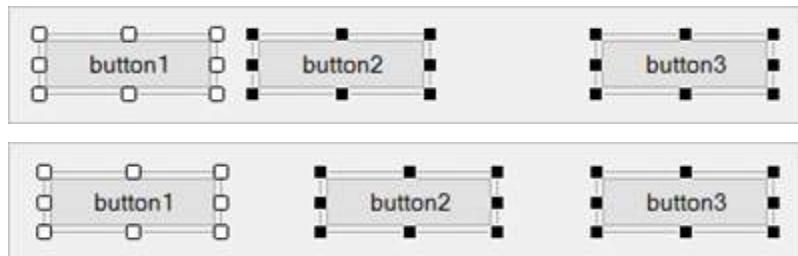
## Control Spacing

The Spacing section of the Designer ribbon has two options to make the distances between the selected controls equal:



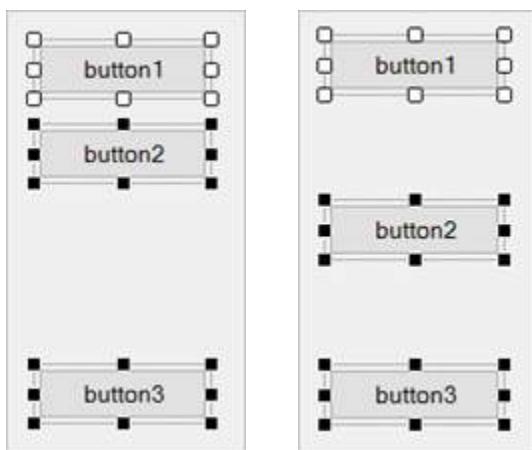
To make the horizontal distances between the selected controls equal

Select Equalize Horizontal Space :



To make the vertical distances between the selected controls equal

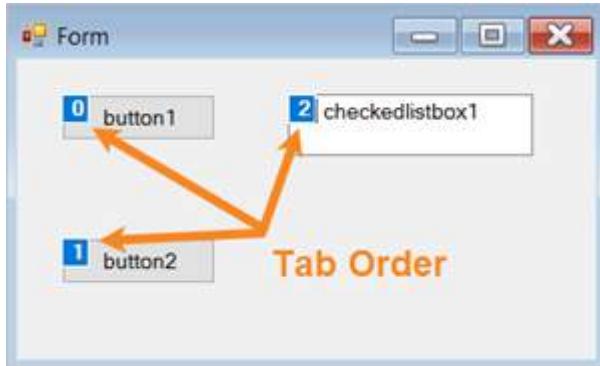
Select Equalize Vertical Space :



## Tab Order

Tab Order works to help you set the order in which you can tab through the form elements. This means that when you press the **Tab** key in a form, each element will be highlighted in a certain order.

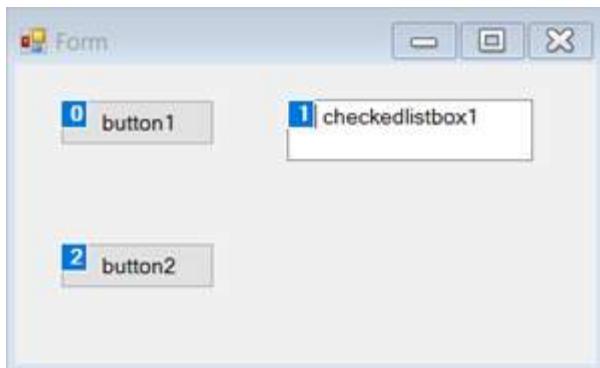
The Tab Order button  on the Designer ribbon is a toggle—one click turns it on and another turns it off. Clicking **Tab Order** will display the tab order of the form items on the top-left of each element.



The tab order of the form elements is a zero-based array. Zero is the first tab, one is the second tab, etc. To change the tab order, click anywhere on an element to incrementally cycle through the available tab orders. When the last available tab order is reached, the numbering starts back at zero.

In the image above the tab order is button1, button2, checkedlistbox1.

In the image below the tab order has been changed to button1, checkedlistbox1, button2.



## Control Property Smart Tags

As mentioned in the [Forms Designer Introduction topic](#), you can edit the properties of the currently selected control in the Properties panel. Some controls support another way of modifying control properties called *smart tags*. These appear as a small icon on the top-right of a control when it is selected in the forms designer.

This image shows the smart tag on a ComboBox control:



Clicking on the smart tag will display options specific to the control which provides an alternative, task-oriented way of modifying control properties:



## 7.4 Preview GUI

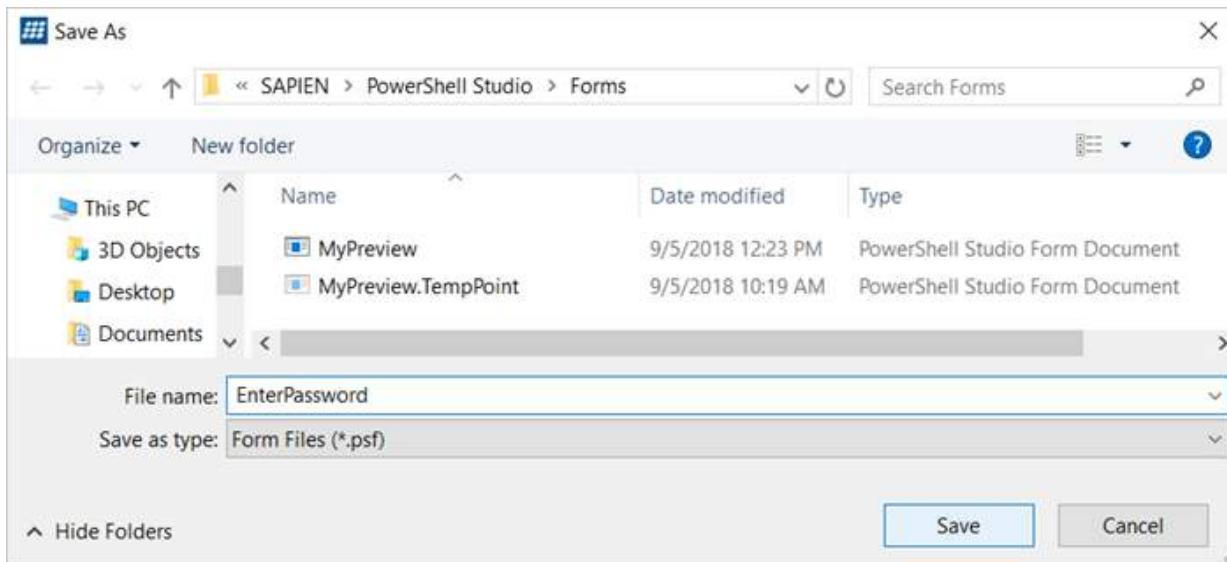
The Preview GUI button allows you to preview the way a form will appear at run time without executing any of your code. None of the controls will work when a form is in preview mode, but you can use the minimize and maximize buttons and also resize the form to make sure it behaves as expected. When you are finished previewing the form, close the form to go back to the Designer.

### To use Preview GUI

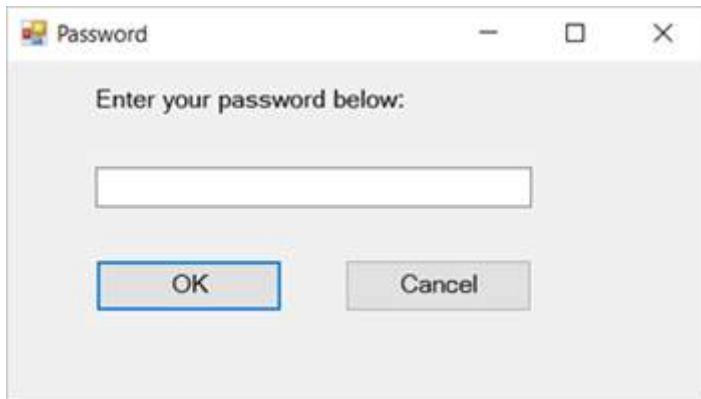
1. Click **Preview GUI** on the Designer ribbon (*Ctrl+Shift+F5*):



2. The **Save As** dialog will display if you have not yet saved the file. Enter a name for the file and click **Save**:



3. The form is displayed:

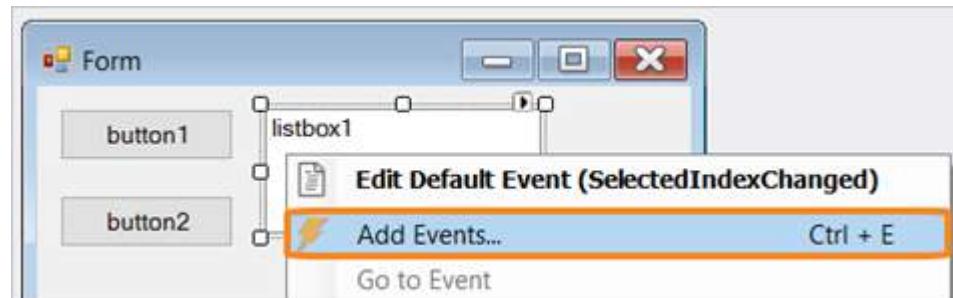


## 7.5 Adding Events

The Add Events dialog is used to connect a control event to your code.

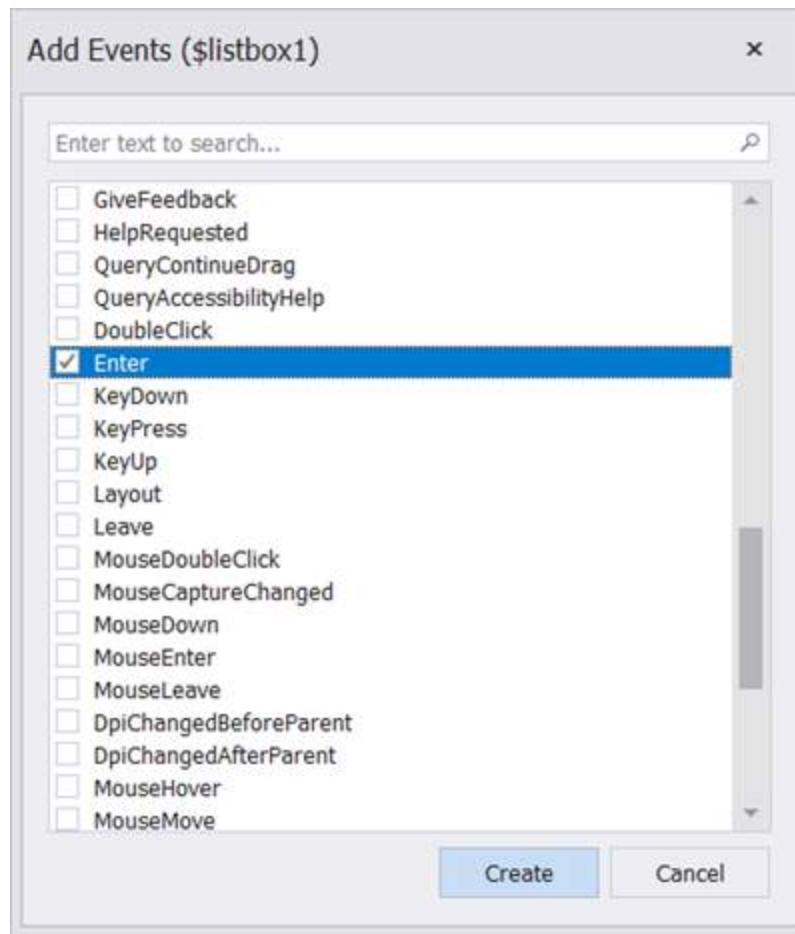
### To add an event

Right-click on the control and choose **Add Events** (*Ctrl+E*):

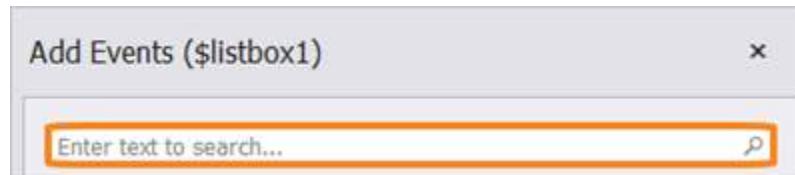


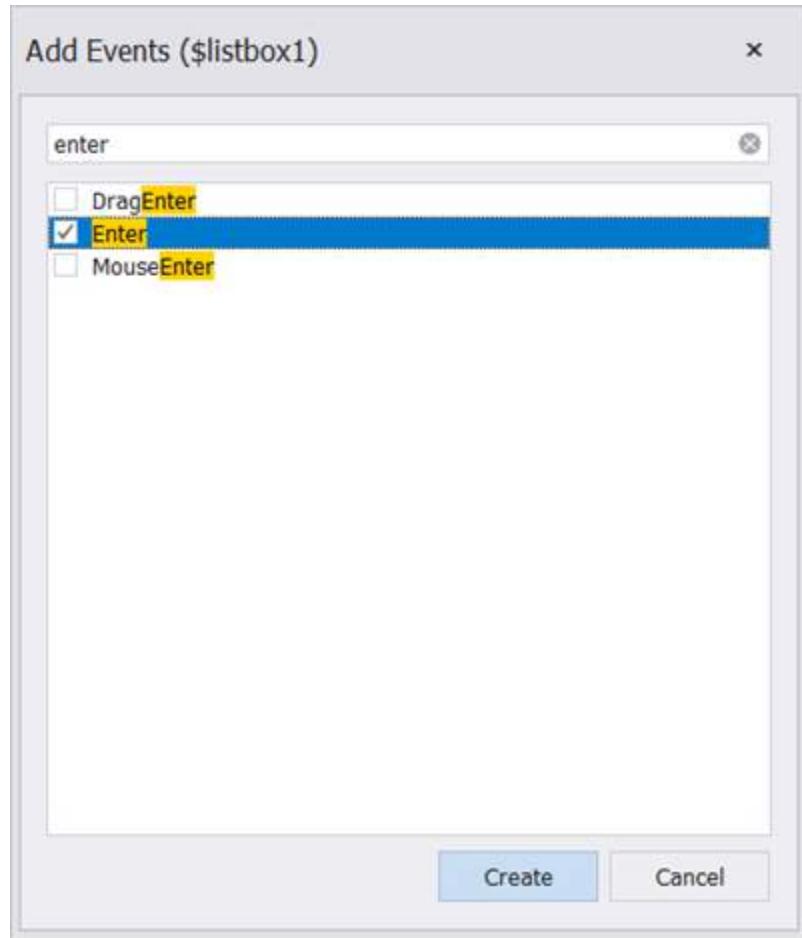
The Add Events dialog launches:

Select one or more event in the Add Events dialog, and then click the **Create** button:



- 👉 Perform a text search to filter the list of events and find the event you are looking for:

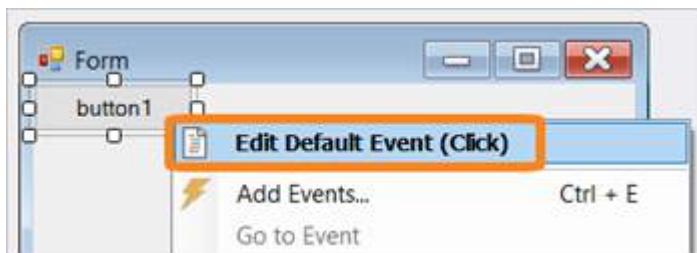




PowerShell Studio will add the required code in the script and indicate where you need to add your own code:

```
69    if ($Items -is [System.Windows.Forms.ListBox+ObjectCollection])
70    {
71        $ListBox.Items.AddRange($Items)
72    }
73    elseif ($Items -is [System.Collections.IEnumerable])
74    {
75        $ListBox.BeginUpdate()
76        foreach ($obj in $Items)
77        {
78            $ListBox.Items.Add($obj)
79        }
80        $ListBox.EndUpdate()
81    }
82    else
83    {
84        $ListBox.Items.Add($Items)
85    }
86
87    $ListBox.DisplayMember = $DisplayMember
88    $ListBox.ValueMember = $ValueMember
89}
#endregion
$listbox1 Enter={
    #TODO: Place custom script here
}
}
```

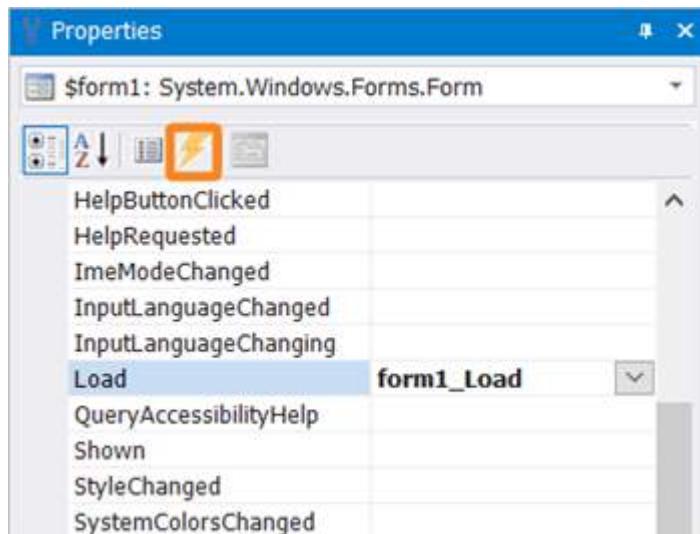
- If the control has a default event, right-click and select **Edit Default Event (<event>)** to automatically add the event code block to the script:



The default control event is added to the script, and the script editor will open with the caret at the position where you can add your own code:

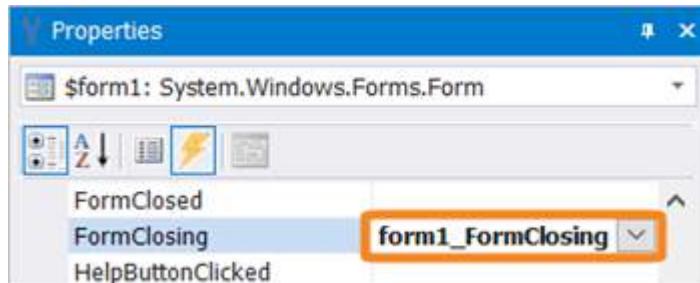
```
1
2 $form1_Load={
3     #TODO: Initialize Form Controls here
4
5 }
6
7 $button1_Click={
8     #TODO: Place custom script here
9
10}
11
```

Event handlers can also be connected to a form or control from the Properties panel. To do this, first select the form or control in the Designer. Then, in the Properties panel, click the lightning bolt button to display the assigned events. The image below shows the events for a form. The *Load* event has been connected to a handler called *form1\_Load*:



To handle another event, simply double-click in the blank cell next to the event. PowerShell Studio will create an event handler named \$<object name>\_<event name>.

For example, double-clicking in the blank cell to the right of the *FormClosing* event connects it to a handler called *form1\_FormClosing*:



The following code is added to the script:

```
98 $form1_FormClosing=[System.Windows.Forms.FormClosingEventHandler]{  
99 #Event Argument: $_ = [System.Windows.Forms.FormClosingEventArgs]  
100     #TODO: Place custom script here  
101  
102 }  
103
```

👉 You can also type the name of the event handler rather than double-clicking in the blank cell. PowerShell Studio will use the name you type to generate the event handler.

## 7.6 Form Templates

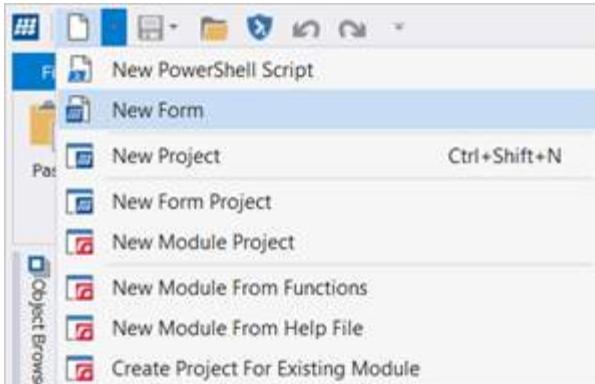
A form template is a predefined form containing controls and script code. Templates help you create GUI scripts quickly by doing much of the layout and code writing for you. This section shows you how to use a predefined template, including grid templates, and also how to create your own templates.

### 7.6.1 Using Predefined Form Templates

This topic shows you how to use a predefined template to create a form.

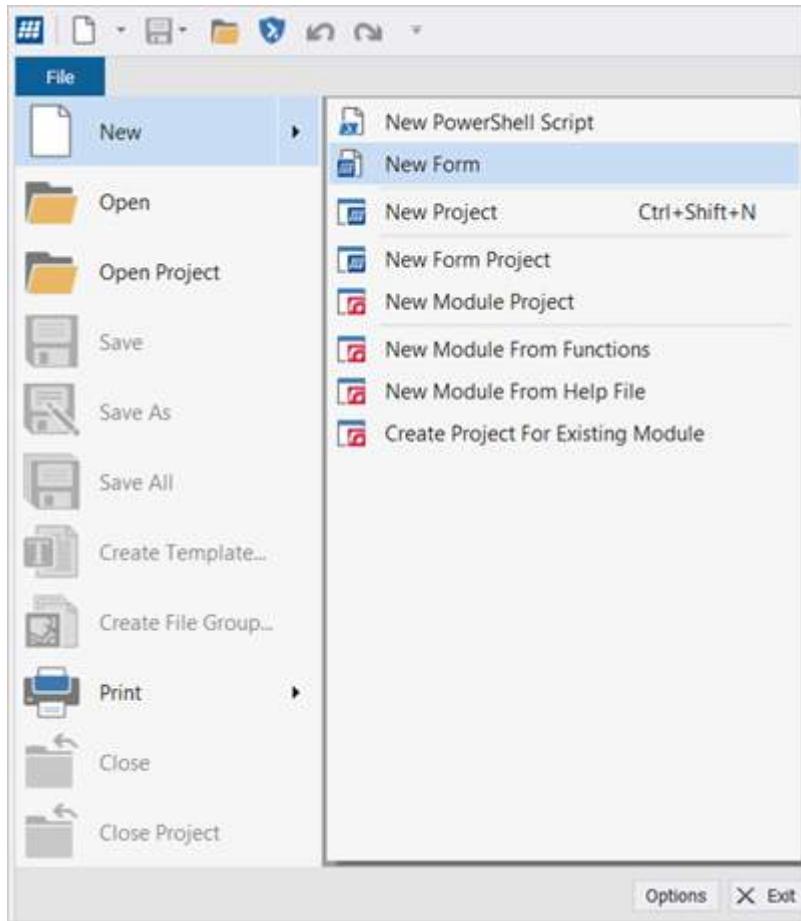
#### To create a GUI based script from a template

- From the Quick Access menu select New > New Form:



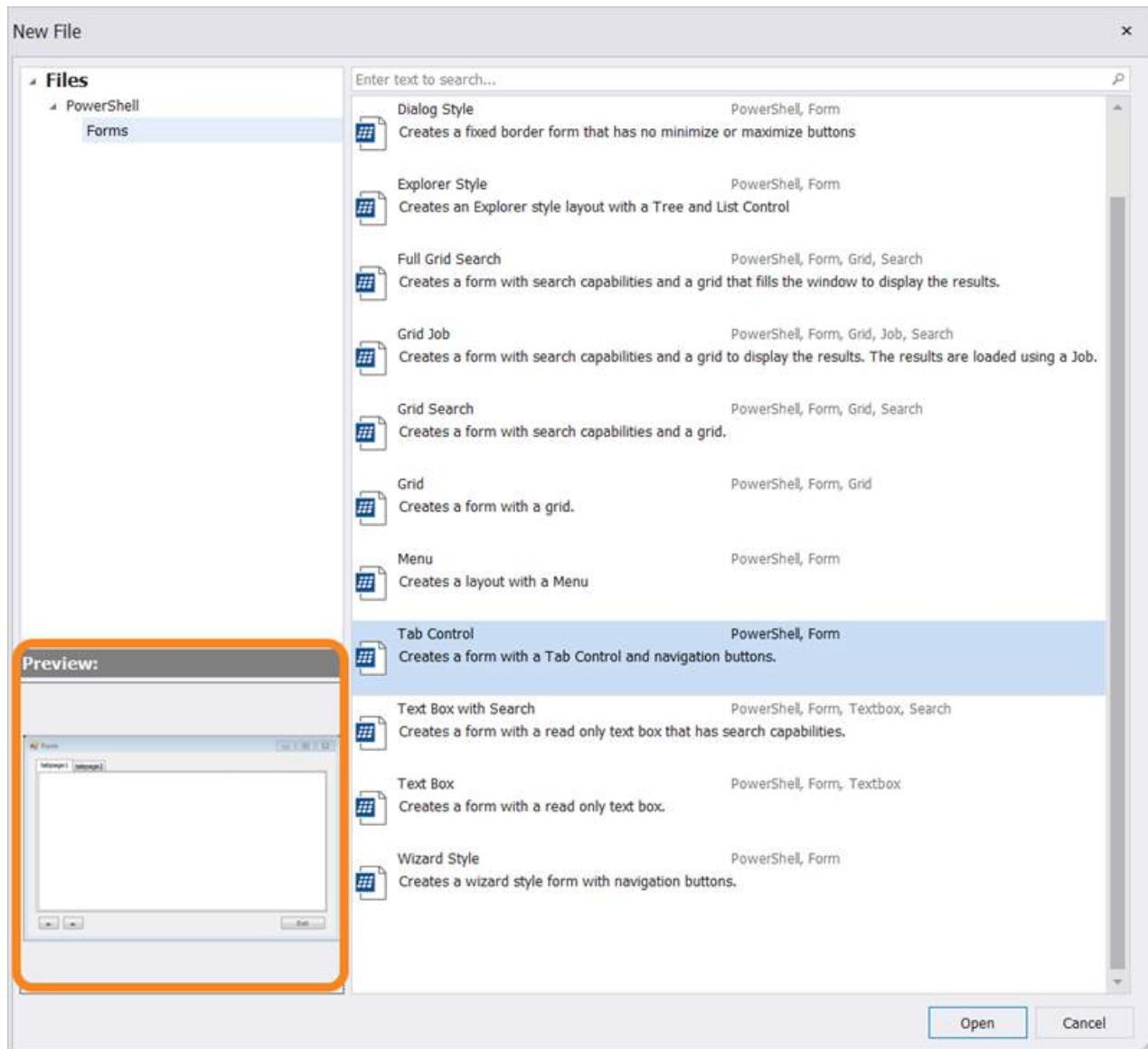
-OR-

- Select File > New > New Form:



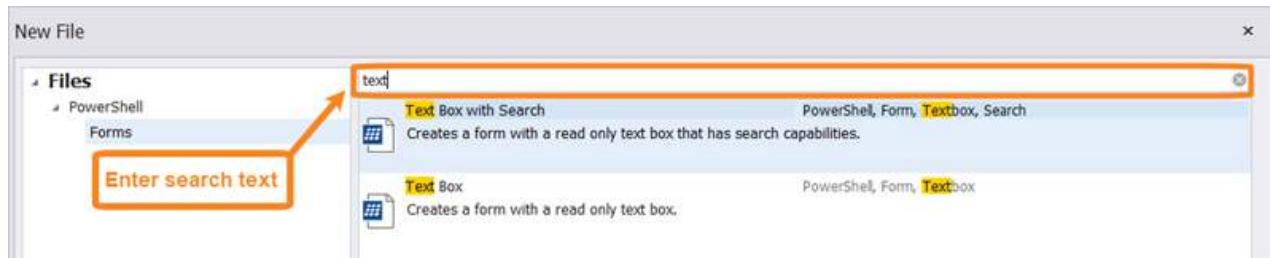
Select a template in the Forms section of the New File dialog, and then click **Open**:

- ➊ When you click on a template a small preview is displayed on the bottom-left. In the image below, the Tab Control form template is selected and the preview is displayed.



PowerShell Studio will create the form and associated scripts, and the form will open in the Designer window of the main editor.

👉 To search for a particular form, type the search criteria in the search field at the top of the dialog and then press <Enter> to search:



### 7.6.2 Creating New Form or Grid Templates

If the predefined templates in PowerShell Studio do not meet your needs, you can create your own.

There are two distinct types of templates:

- **Form templates**

A form template can be based on any existing form. It is a general purpose template.

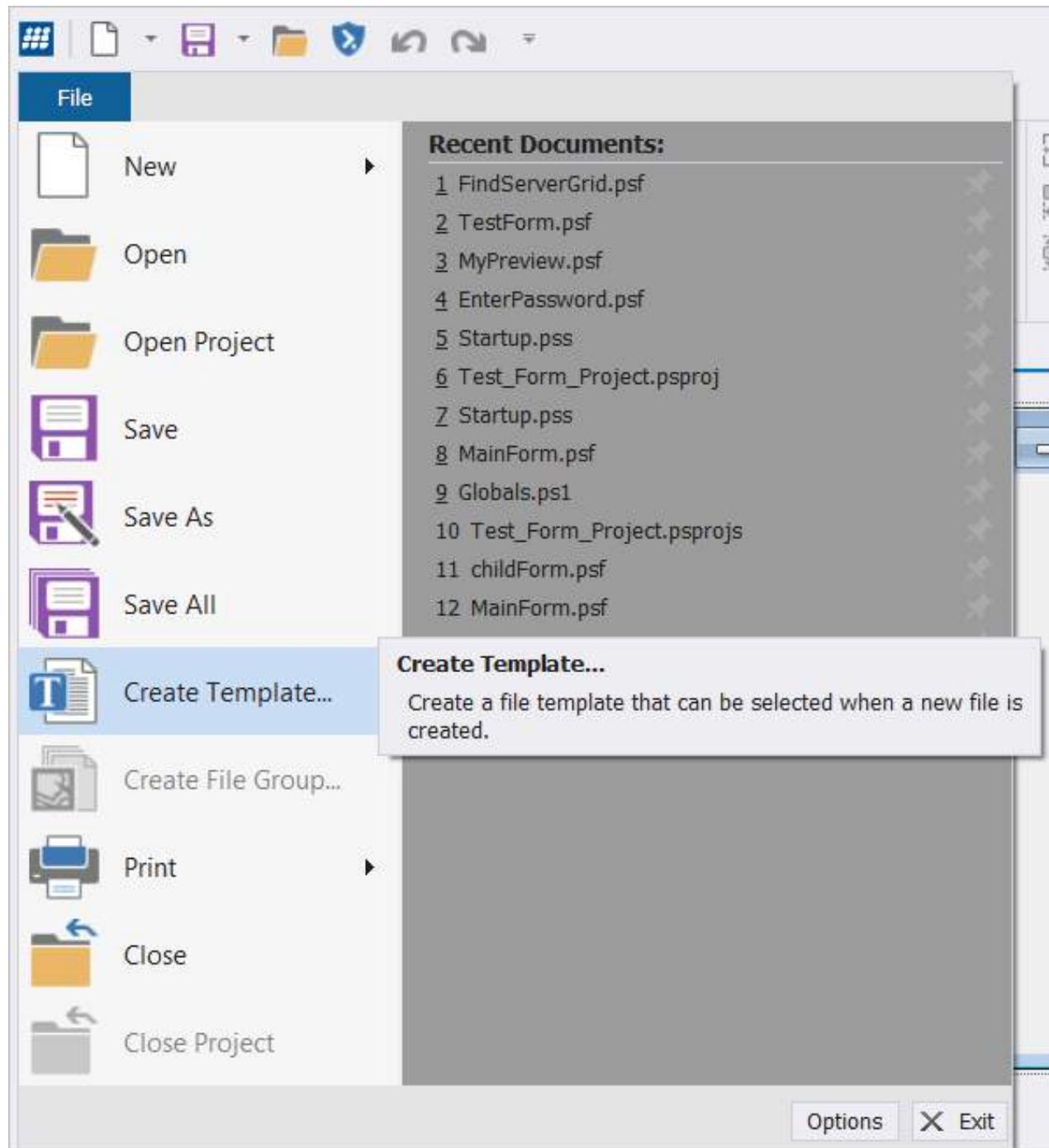
- **Grid templates**

[Grid templates](#) can only be created from an existing form that contains a DataGridView control.

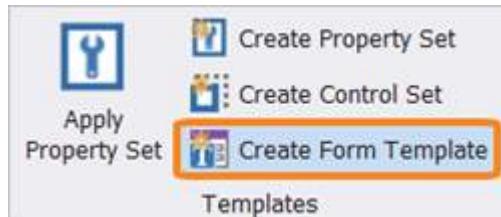
These templates are designed to be used in conjunction with the Object Browser to rapidly create forms that retrieve database records or WMI objects and display them in a grid.

To get started, create a new form and configure it as required including event handlers and any other code.

Save your work and then from the **File** menu, choose **Create Template**:



You can also click the **Create Form Template** button in the Templates section on the Designer ribbon:



The Create File Template dialog will open:



Fill in the template properties:

- **Tags**

PowerShell Studio will prepopulate appropriate tags/keywords that describe the contents of the template. You can add additional comma-separated tags.

- **Type**

PowerShell Studio supports two types of templates:

- **File (form)**

- **Grid**

If your form does not include a DataGridView control this option will not be available.

👉 You can [create a form template with a DataGridView control](#) by right-clicking an object in the [Database Browser](#) or [WMI Browser](#) and selecting **Generate Query Form....**

- **Creator**

Your name. (The value for this field is taken from **Home > Options > General > Username**).

- **Company**

Your company details. (The value for this field is taken from **Home > Options > General > Organization**).

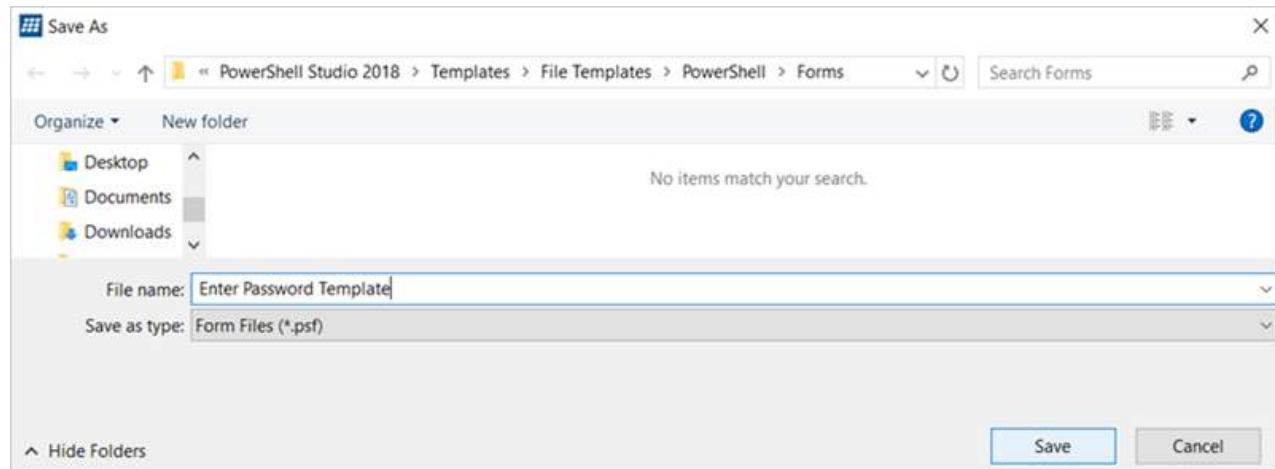
- **Description**

A description of the purpose of the template.

- **Target Grid**

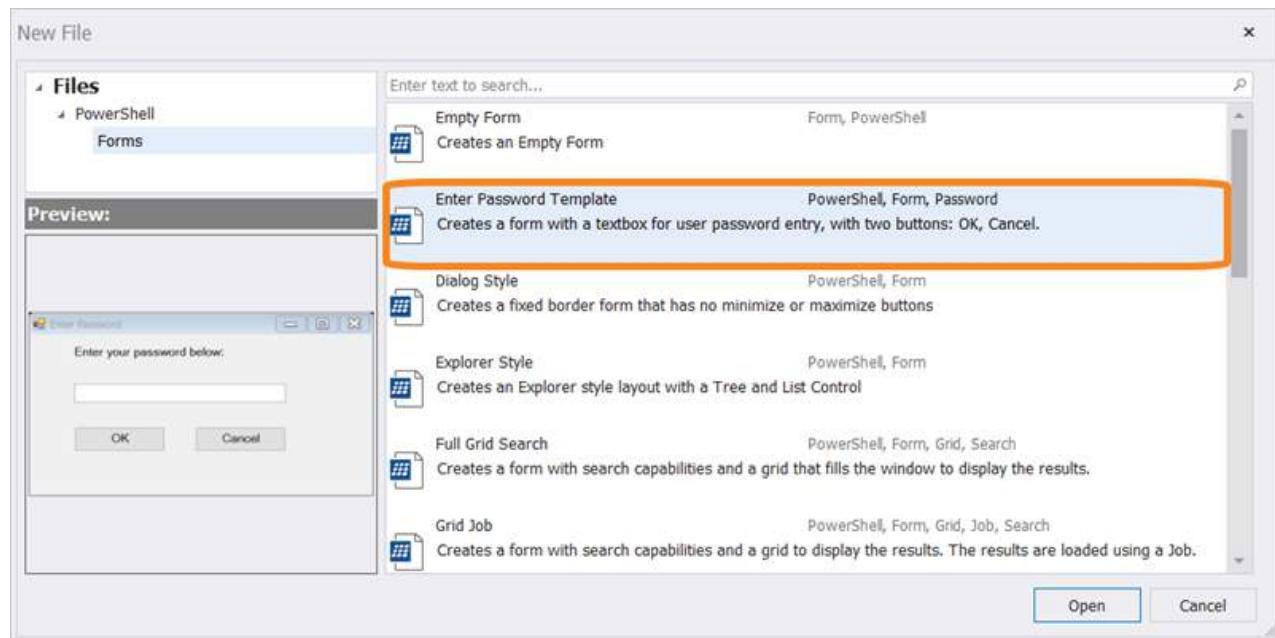
This option is available if you are creating a grid template. In cases where a form has more than one grid, it allows you to specify which grid auto generated code will use.

Enter a name for the template and then click **Save**:



**i** The default location for user created templates is `%AppData%\SAPIEN\PowerShell Studio\Templates\File Templates\PowerShell\Forms`. The default location can be changed in **Home > Options > General > Directories > Template Directory**.

The new template will appear in the list of standard templates whenever you create a new form:

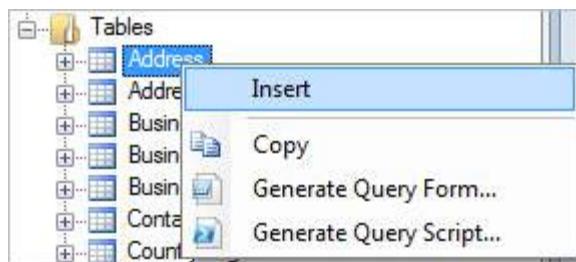


**👍** Add *Tags* when you create the form template so that you can easily find it with a text search.

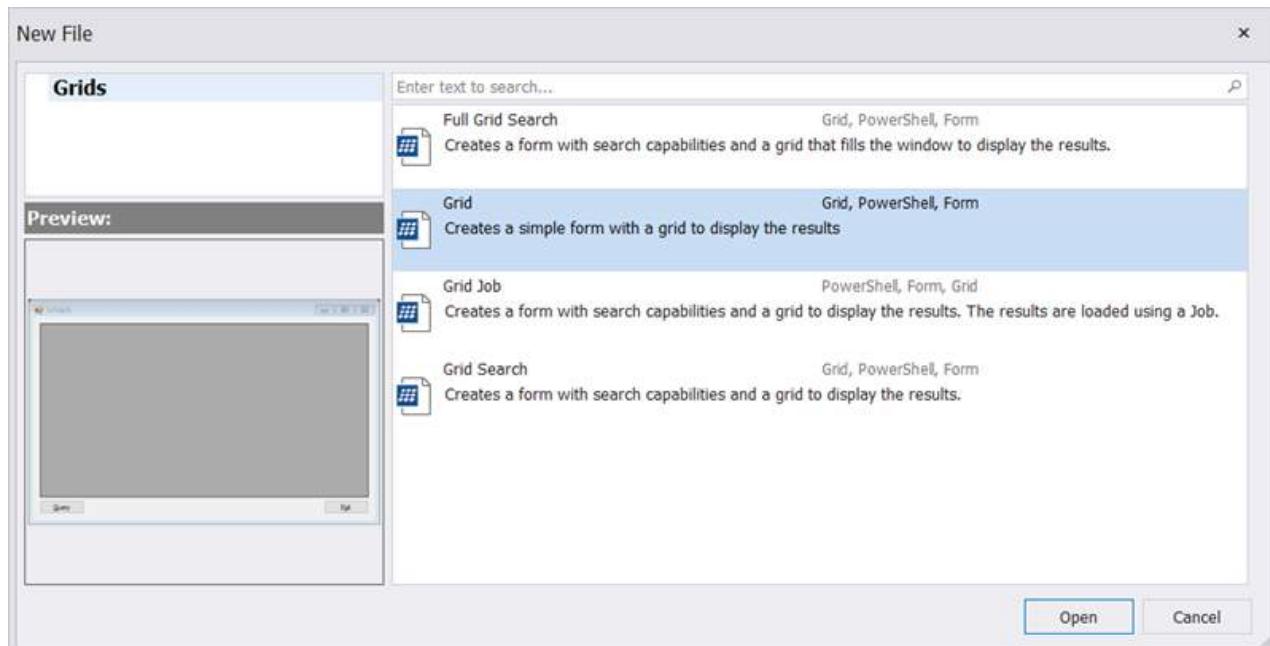
### 7.6.3 Working with Grid Templates

This topic explains how to work with grid templates.

If you right-click on a database table or WMI object in the Object Browser, the following menu appears:



The **Generate Query Form...** option will display a list of grid templates that include code to retrieve objects and display them in a grid:



Placeholders are included in the template, allowing the code generator to reuse a template for different kinds of objects. These placeholders get replaced with object specific code when the template is used to create a grid form.

There are two placeholders:

- **%ResultsFunction%**

This placeholder is replaced by a function declaration that returns the results of a query.

- **%ResultsFunctionCall%**

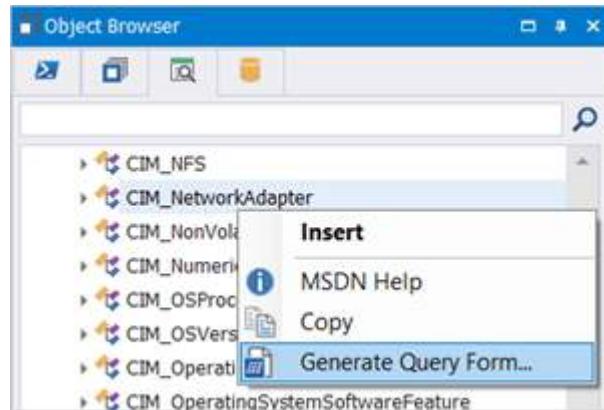
This placeholder is replaced by a call to the results function.

If you open the form file 'Grid.psf' from the built in grid template (%ProgramData%\SAPIEN\PowerShell Studio <year>\Templates\Grid Templates) and examine the code, you will see the two 'results' placeholders:

```
1 #region Control Helper Functions
206
207 $formMain_Load={
208     #TODO: Initialize Form Controls here
209
210 }
211
212 %ResultsFunction%
213
214 $buttonExit_Click={
215     #TODO: Place custom script here
216     $formMain.Close()
217 }
218
219 $buttonQuery_Click={
220     #TODO: Place custom script here
221     $results = %ResultsFunctionCall%
222     $results = ConvertTo-DataTable -InputObject $results -FilterWMIProperties
223     Update-DataGridView -DataGridView $datagridviewResults -Item $results -AutoSizeColumnsMode DisplayedCells
224 }
225
226
227
228 $datagridviewResults_ColumnHeaderMouseClick=[System.Windows.Forms.DataGridViewCellEventHandler](...)
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
```

PowerShell Studio will create the code for the placeholders when you generate a grid template against a WMI object.

For example, right-click the *CIM\_NetworkAdapter* object in the WMI Browser and select **Generate Query Form**:



Next select the **Grid** template, click **Open**, and PowerShell Studio will generate the grid form with the following code:

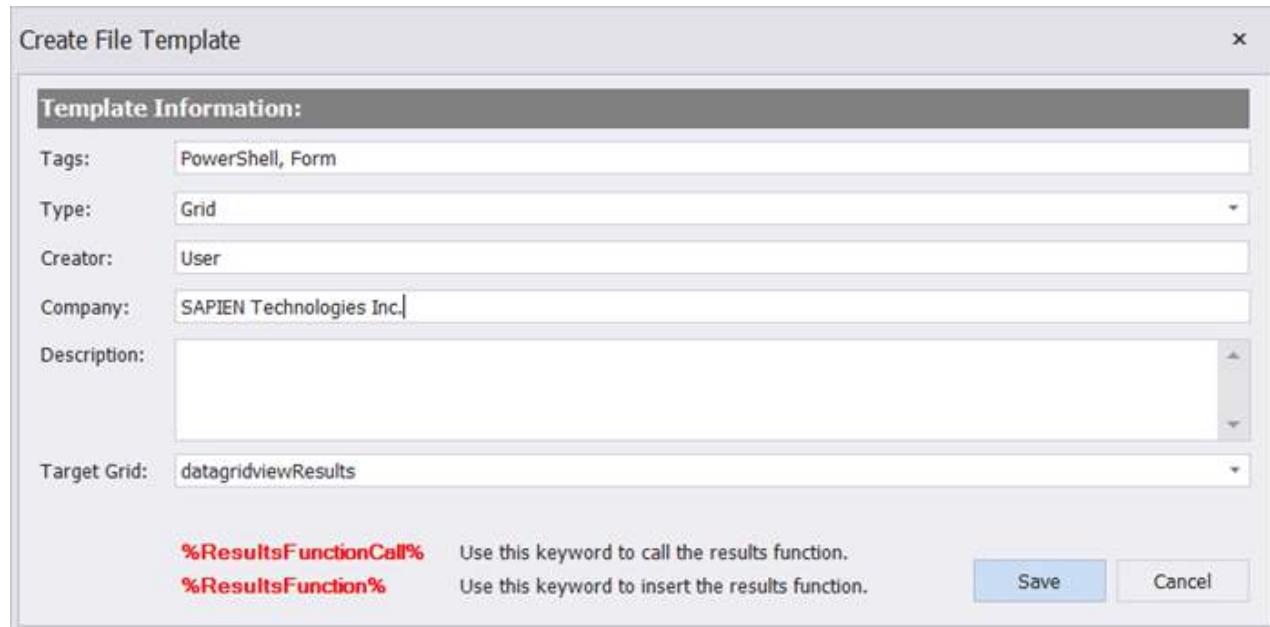
```

1 #region Control Helper Functions
206
207 $formMain_Load={
208     #TODO: Initialize Form Controls here
209 }
210 }
211
212 function Get-WMI_CIMV2-CIM_NetworkAdapter
213 {
214     #Run the WMI Query
215     $results = Get-WmiObject CIM_NetworkAdapter -Namespace "Root\CIMV2"
216     return $results
217 }
218
219 $buttonExit_Click={
220     #TODO: Place custom script here
221     $formMain.Close()
222 }
223
224 $buttonQuery_Click={
225     #TODO: Place custom script here
226     $results = Get-WMI_CIMV2-CIM_NetworkAdapter
227     $results = ConvertTo-DataTable -InputObject $results -FilterWMIProperties
228     Update-DataGridView -DataGridView $datagridviewResults -Item $results -AutoSizeColumnsMode DisplayedCells
229 }
230
231 $datagridviewResults_ColumnHeaderMouseClick=[System.Windows.Forms.DataGridViewCellEventHandler][...]
246

```

These placeholders allow you to control exactly how a user can interact with data, while leaving PowerShell Studio to write the data retrieval code.

When you create a grid template from an existing form, PowerShell Studio will indicate if you have not included the 'results' placeholders in your code:



You can elect not to include them if you would rather provide a fixed data retrieval implementation.

## 7.7 Exporting Form Scripts

You can export a script directly from the Deploy tab on the ribbon, in the **Export** section:



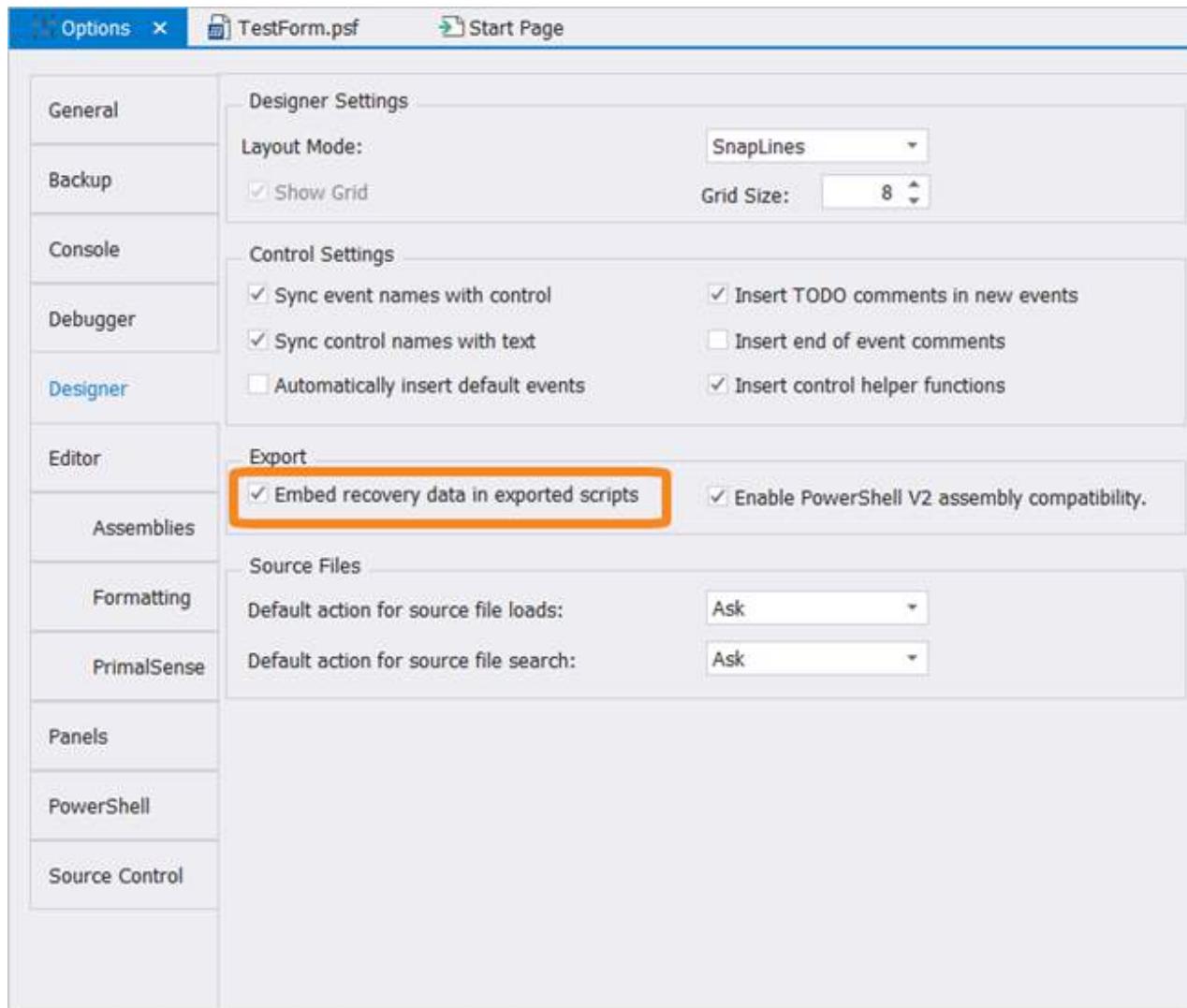
There are two export options:

- **Export to File** - Exports the script to a ps1 file
- **Export to Clipboard** - Copies the script to the clipboard.

Exporting creates a single stand-alone script that encapsulates all of the content of a script or package. The exported code can be placed on the clipboard or stored in a file. In both cases, PowerShell Studio adds assembly load statements to the front of the exported code to make sure that it runs in the same environment.

The PowerShell Studio export process can also add metadata, called recovery data, to the script. This metadata allows PowerShell Studio to recreate the original project that was used during the export. The recovery data is stored in multi-line comment blocks in the exported script.

Recovery data embedding is enabled by default (**Home > Options > Designer > Embed recovery data in exported scripts**):



For information on the recovery options, see [Options and Settings > Designer > Designer Settings](#) [343]

- When an exported script is opened in PowerShell Studio and recovery data is present, PowerShell Studio will offer you the option of using the recovery data to recreate the project that was used to create the script. Alternatively, you can open the script in the code editor.

## 7.8 Initializing GUI Controls

The Form control's **Load** event is a convenient place to initialize GUI controls because it is called right before the form is displayed:

```
$formMain_Load={  
    #TODO: Initialize Form Controls here  
}  
  
function Get-WMI_Appv-AppvClientApplication  
{  
    #Run the WMI Query  
    $results = Get-WmiObject AppvClientApplication -Namespace "Root\appv"  
    return $results  
}  
  
$buttonExit_Click={  
    #TODO: Place custom script here  
    $formMain.Close()  
}
```

! Do not try to initialize a control outside an event block within the script:

- The control might not exist when this line is executed by PowerShell.
- If the control does exist, your changes will most likely get overwritten by the designer generated script.

## Alternatives to the Load Event

It is possible to use other events to trigger initialization, such as the *VisibleChanged* event in PowerShell Studio's *Chart - Disk Space* control set. This event is triggered using the *Visible* property—when the control is displayed (i.e., loaded), or when the control is hidden.

```
$chartDiskSpace_VisibleChanged={  
    if($this.Visible)  
    {  
        Update-DiskChart $this  
    }  
}
```

In the example above, the control set initializes when the control becomes visible by checking its *Visible* property.

If your initialization script is running slow, it can prevent the GUI from displaying in a timely manner or cause it to hang. In instances such as this, you might want to delay your initialization or simply run the query in a separate job, then initialize and enable the controls after the job has completed the query.

- i** You cannot access GUI controls directly from a Job. You must return the results first and then initialize the controls on the main thread / script.

## 7.9 Control Helper Functions

This topic explains control 'helper functions' and their rules, and also shows you how to add custom helper functions.

### About Control Helper Functions

Some controls are relatively complicated to work with directly from PowerShell, and may require custom .Net code to be able to access all of their features. For those controls, PowerShell Studio automatically creates helper functions that add useful .Net based methods into the form.

For example, when you add a ListView control to a form, PowerShell Studio adds two helper functions:

#### 1. Update-ListViewColumnSort

This function enables sorting on any of the ListView's columns.

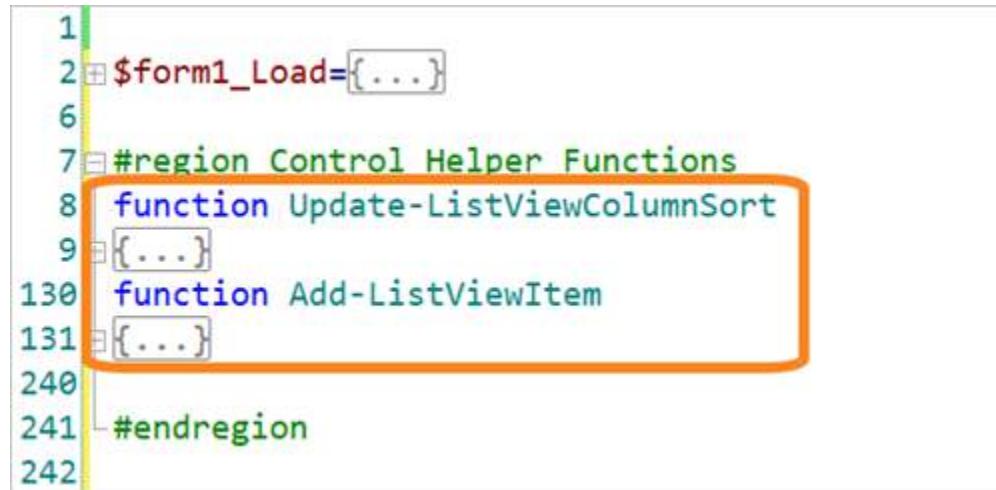
#### 2. Add\_ListViewItem

This function provides an easy way to add items to a ListView

To demonstrate this, create a new empty form in PowerShell Studio and look at the script that has been created:

```
1
2 $form1_Load={
3     #TODO: Initialize Form Controls here
4
5 }
6
```

Now add a ListView control to the form and examine the code again. In the image below the helper functions have been folded to reduce the size of the image:



```
1
2 $form1_Load=[...]
6
7 #region Control Helper Functions
8 function Update-ListViewColumnSort
9 [...]
130 function Add-ListViewItem
131 [...]
240
241 #endregion
242
```

A region called 'Control Helper Functions' has been added to the form and two functions have been defined.

## Adding Custom Helper Functions

As you build larger and more complex scripts, you will find it useful to build up a collection of helper functions. Each control that has helper functions will have a folder named after the control. Each of these directories contains one or more PowerShell scripts that will be merged into existing code when the control is added to a form.

This is the default set of helper functions delivered with PowerShell Studio:

- System.Windows.Forms.CheckedListBox
- System.Windows.Forms.ComboBox
- System.Windows.Forms.DataGridView
- System.Windows.Forms.DataVisualization.Charting.Chart
- System.Windows.Forms.Integration.ElementHost
- System.Windows.Forms.ListBox
- System.Windows.Forms.ListView
- System.Windows.Forms.NotifyIcon
- System.Windows.Forms.ToolStripComboBox
- System.Windows.Forms.TreeView

- i** The default location for control functions is %ProgramData%\SAPIEN\PowerShell Studio <year>\Templates\Control Functions.

In the example below we will add a helper function to textbox controls that converts the text in a textbox to uppercase.

First we create a folder called **System.Windows.Forms.TextBox** in the default folder location shown above. The folder name must match the control type's full name:

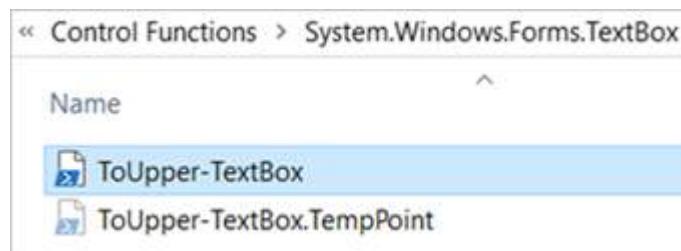
- System.Windows.Forms.CheckedListBox
- System.Windows.Forms.ComboBox
- System.Windows.Forms.DataGridView
- System.Windows.Forms.DataVisualization.Charting.Chart
- System.Windows.Forms.Integration.ElementHost
- System.Windows.Forms.ListBox
- System.Windows.Forms.ListView
- System.Windows.Forms.NotifyIcon
- **System.Windows.Forms.TextBox**
- System.Windows.Forms.ToolStripComboBox
- System.Windows.Forms.TreeView

Next we would create one or more scripts, one for each helper function.

In this example, we will create the script shown below:

```
1 function ToUpper-TextBox
2 {
3 <#
4     .SYNOPSIS
5         This function helps you convert textbox content to uppercase.
6
7     .DESCRIPTION
8         Use this function to convert textbox content to uppercase.
9
10    .PARAMETER ListBox
11        The TextBox control you want to add items to.
12
13    .EXAMPLE
14        ToUpper-TextBox $textbox1
15
16 #>
17
18 param (
19     [ValidateNotNull()]
20     [Parameter(Mandatory = $true)]
21     [System.Windows.Forms.TextBox]$TextBox
22 )
23
24 $TextBox.Text = $TextBox.Text.ToUpper()
25 # position cursor at end of replaced text
26 $TextBox.Select($TextBox.Text.Length, 0)
27 }
28
```

The script file is saved as the helper function name *ToUpper-TextBox* in the **System.Windows.Forms.TextBox** folder:



- i** The name of the function must be the same as the name of the script file, and there should be only one file per script.

Now when we add a textbox to a form in PowerShell Studio, our helper function will be included along with the standard code as shown below:

The screenshot shows a code editor window with the following PowerShell script:

```
1 $form1_Load={
2     #TODO: Initialize Form Controls here
3
4 }
5
6
7 #region Control Helper Functions
8 function ToUpper-TextBox
9 {
10     <#...
11
12     param (
13         [ValidateNotNull()]
14         [Parameter(Mandatory = $true)]
15         [System.Windows.Forms.TextBox]$TextBox
16     )
17
18     $TextBox.Text = $TextBox.Text.ToUpper()
19     # position cursor at end of replaced text
20     $TextBox.Select($TextBox.Text.Length, 0)
21 }
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

A blue rectangular box highlights the region `#region Control Helper Functions` and the function `ToUpper-TextBox`. An orange rounded rectangle highlights the entire function body, from the opening brace to the closing brace.

## Helper Function Rules

- The folder must be named after the control type's full name.
- The file name must match the function name.
- Each file should only contain a single function.

**!** Failure to follow these rules can result in the insertion of multiple copies of the same function.

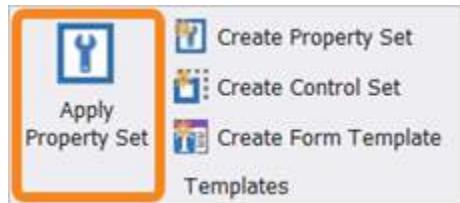
## 7.10 Property Sets

A property set is a collection of property settings that can be applied together to one or more controls. Property sets enable you to quickly create a consistent look and feel to your forms by applying a group of properties to existing controls, such as anchoring, font, and coloring. A property set can be generic, or specific to a particular type of control. This topic shows you how to apply and create property sets.

### Applying Property Sets

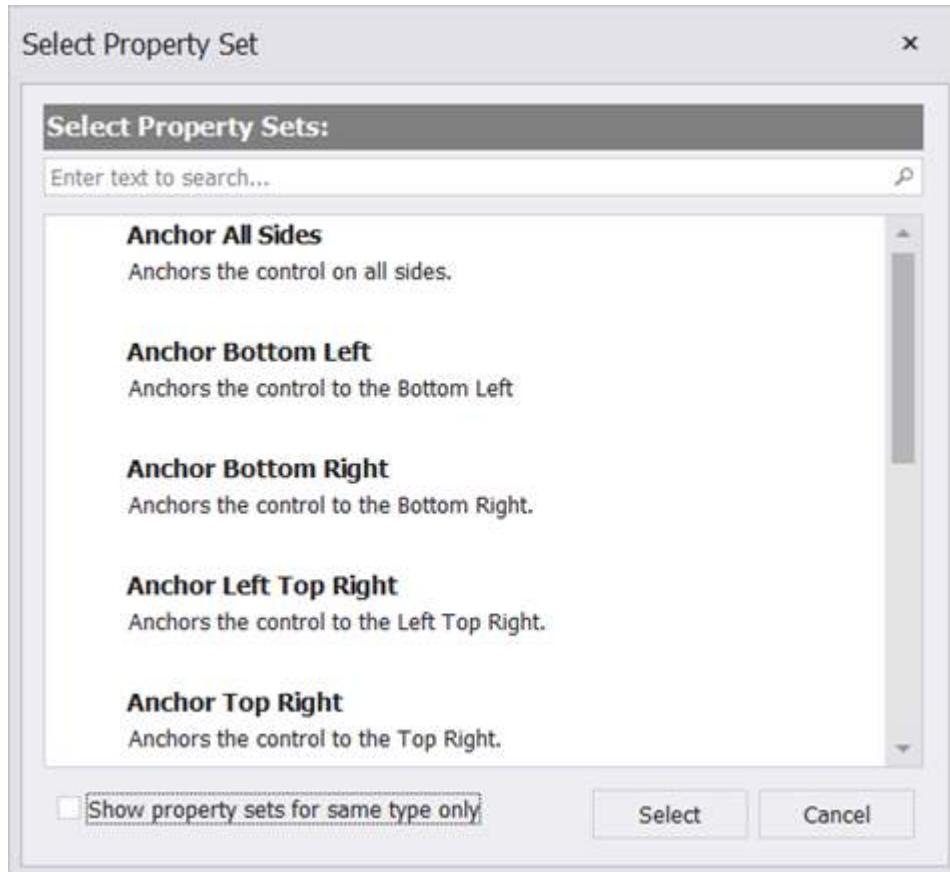
#### To apply a property set

Select one or more controls on a form and then click **Apply Property Set** on the Designer ribbon:



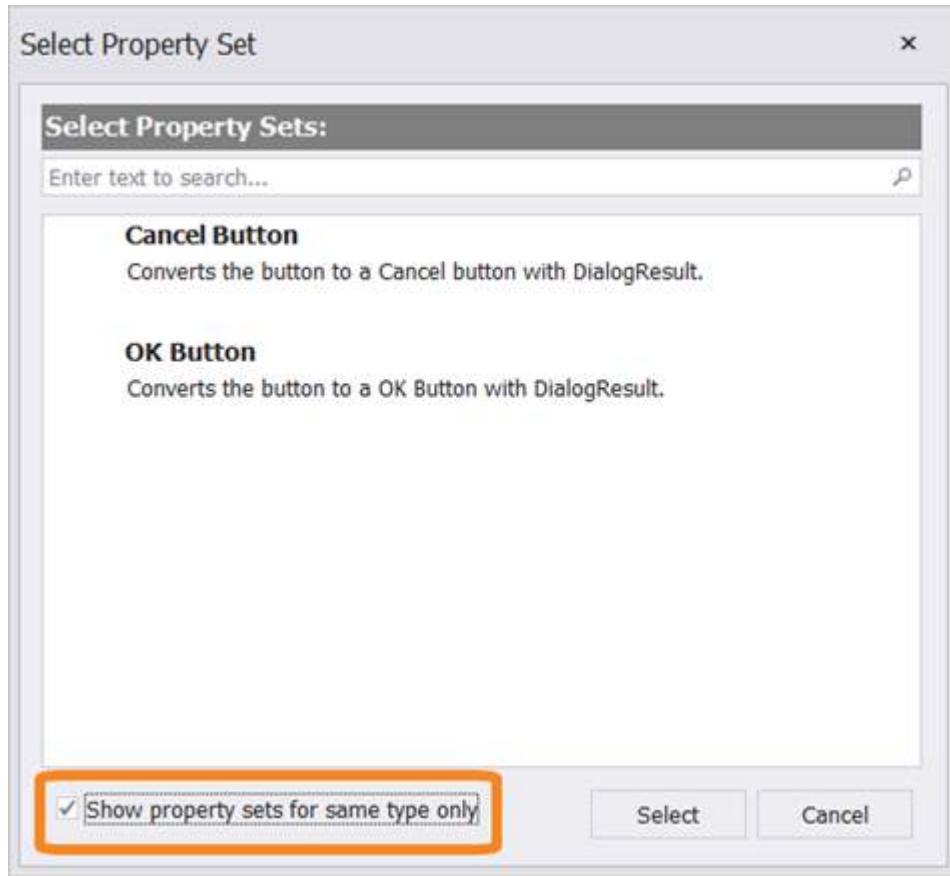
Or you can right-click on a control and select **Apply Property Set** (*Ctrl+L*).

The Select Property Set dialog will appear:



In this example we are applying a property set to a button control. PowerShell Studio provides six general styles and two that are specific to buttons.

Selecting the **Show property sets for same type only** check box will filter the list to show styles that apply directly to the selected control type:



Select a property set and then click **Select** to apply it.

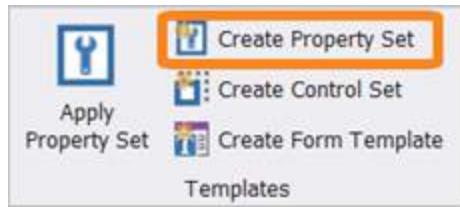
- ⓘ If you modify a property set after you have applied it to controls, you must reapply it to those controls. Your changes will *not* be automatically applied.

## Creating Property Sets

You can create your own property sets that encapsulate local branding requirements or other standard settings.

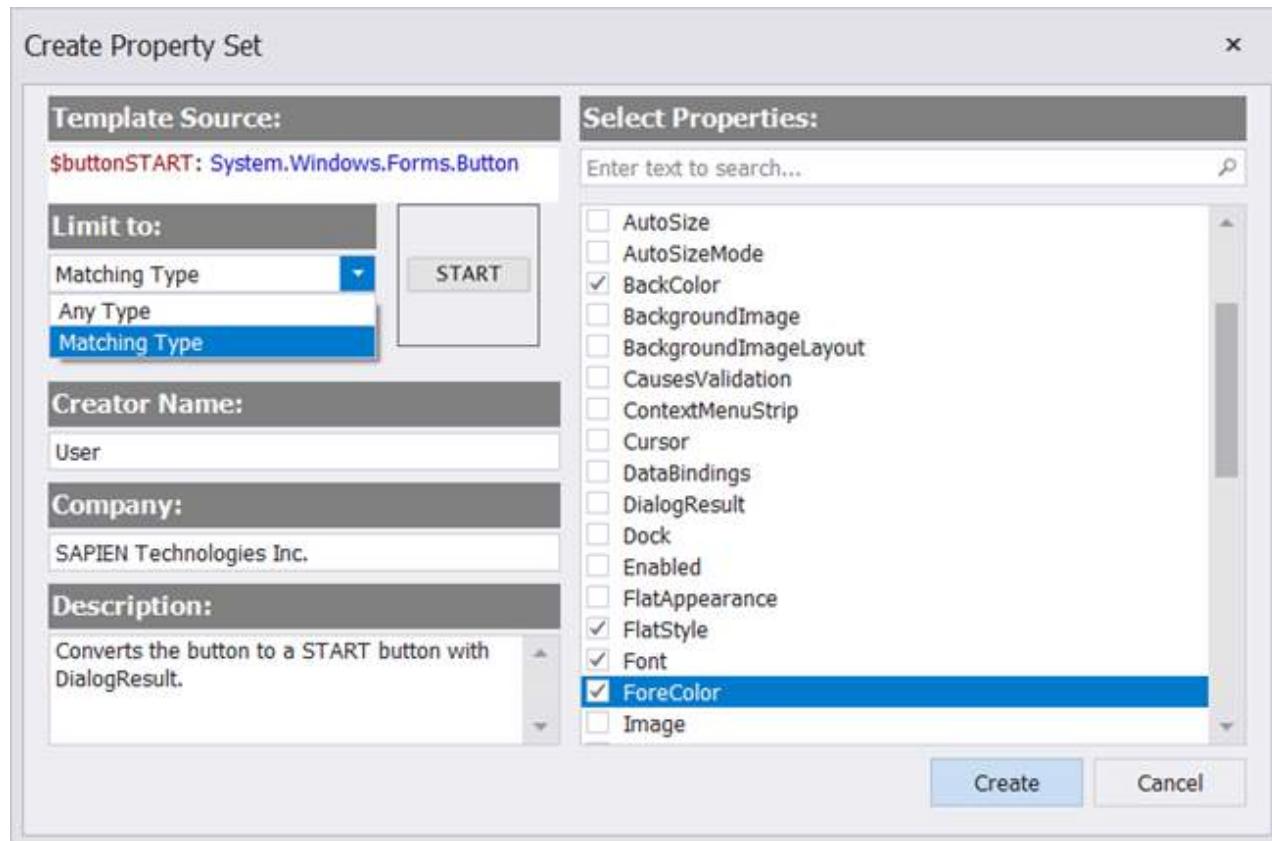
### To create a property set

Configure all of the control properties that you want to capture, then select a control and click **Create Property Set** on the Designer ribbon:



Or you can right-click on a control and select **Create Property Set (Ctrl+Shift+L)**.

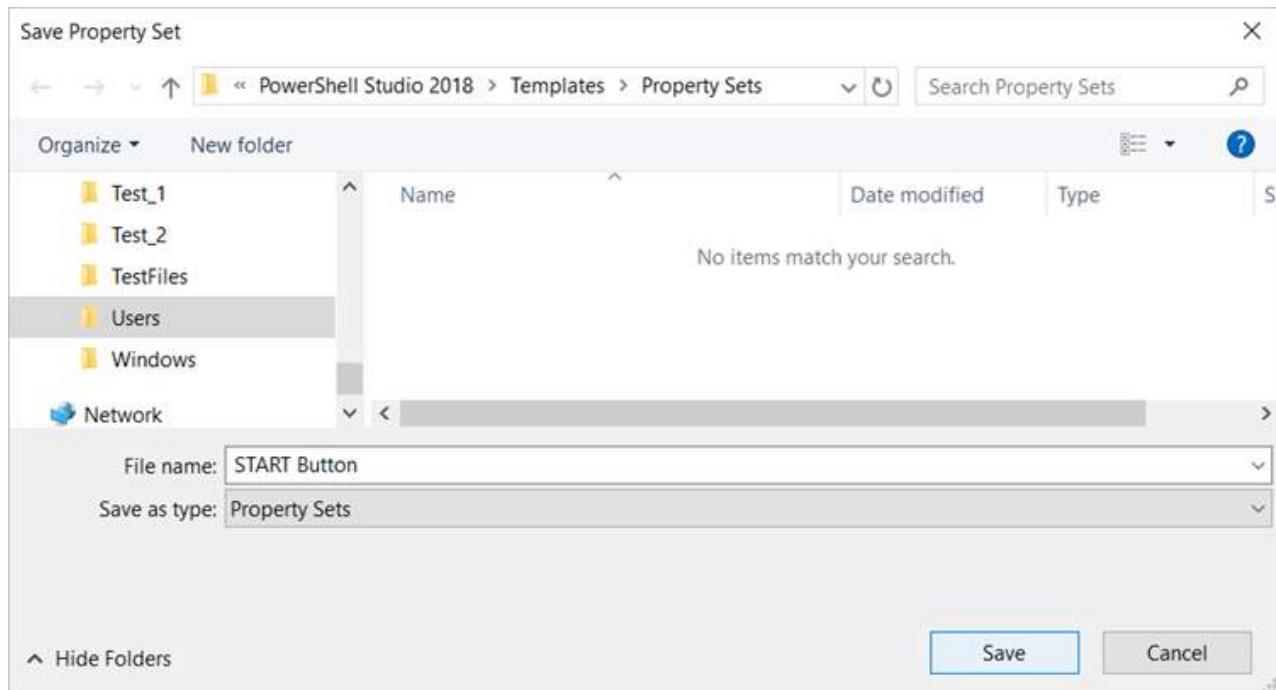
The Create Property Set dialog allows you to configure the new property set:



Configure the options in the Create Property Set dialog and then click **Create**:

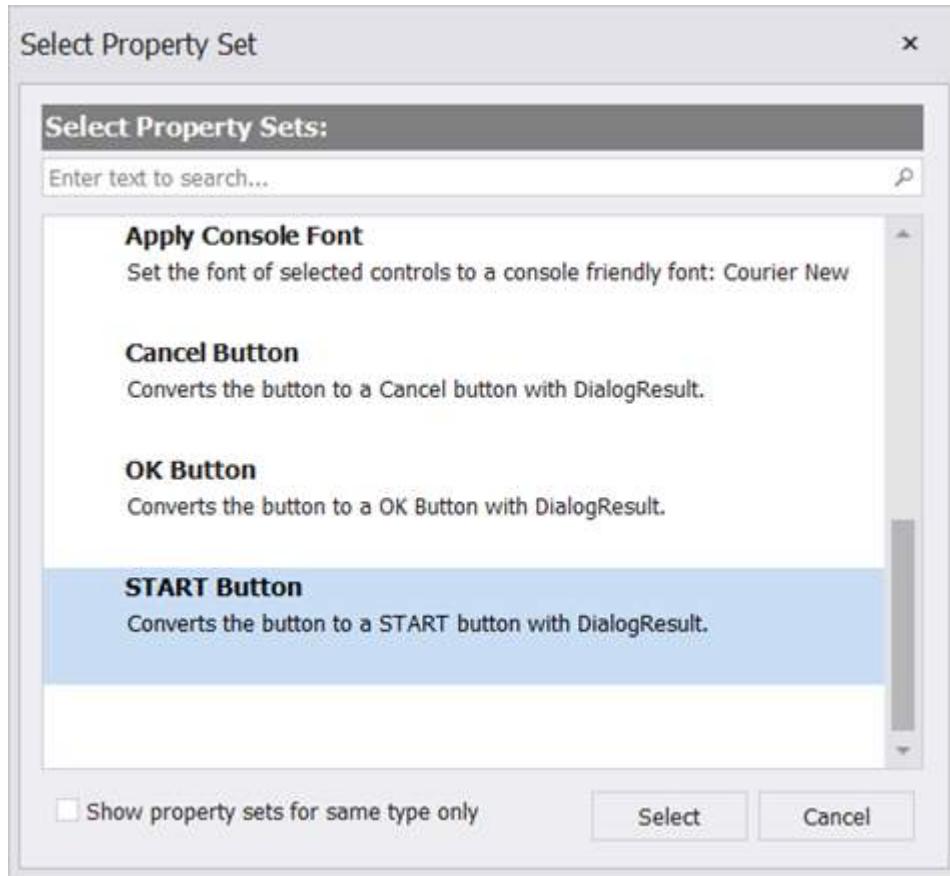
- **Limit to:**
  - a. **Any Type**  
The property set can be used with any type of control.
  - b. **Matching Type**  
The property set can only be used with controls that are the same type as the source control.
- **Creator Name**  
Your name. (The value for this field is taken from **Home > Options > General > Username**).
- **Company**  
Your company details. (The value for this field is taken from **Home > Options > General > Organization**).
- **Description**  
Provide a description of the property set. *This field is mandatory.*
- **Select Properties**  
Select the properties that should be captured by the property set.

Enter a name for the property set and click **Save**:



- i** The default location for custom property sets is %AppData%\SAPIEN\PowerShell Studio <year>\Templates\Property Sets.

The next time you apply a property set, the new property set will appear in the property set list:



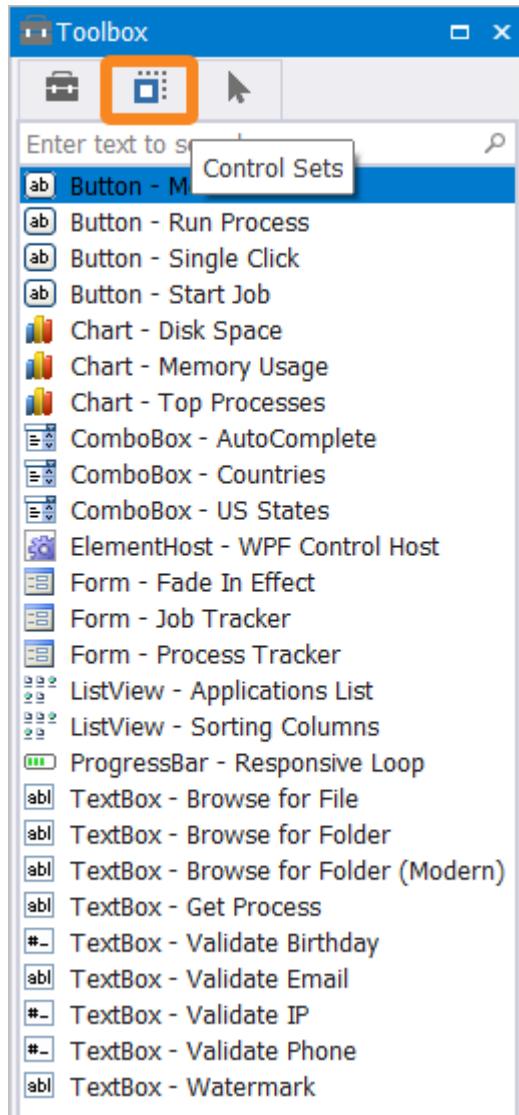
- i** There is currently no way to alter a property set—they must be recreated and reapplied to the control.

## 7.11 Control Sets

In the same way that [property sets](#) allow you to group together property settings for reuse, *control sets* address common PowerShell scripting scenarios by combining multiple controls and script code into new custom controls that can be added to forms just like standard controls. This topic shows you how to insert and create control sets.

### Inserting Control Sets

Control sets are located in the **Control Sets** tab of the Toolbox panel:

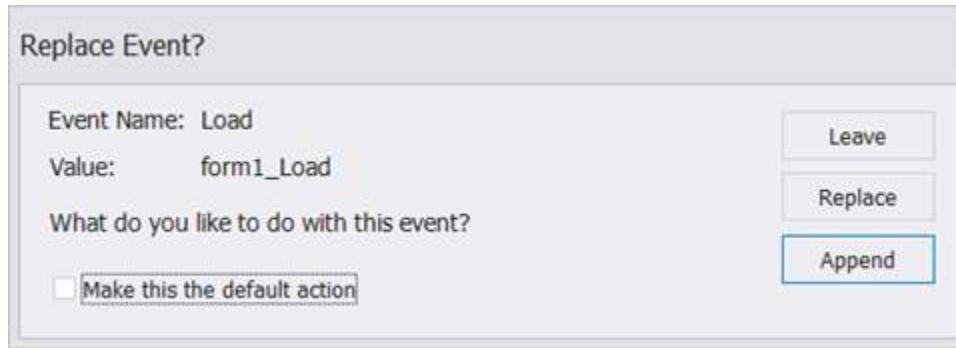


## To add a control set to a form

Use any of these options to add a control set to a form:

- Double-click the control set.
- Drag and drop the control set onto the form.
- Right-click on the control set and select **Insert**.

In some cases, a control set will need to add code to certain event handlers before it can work. PowerShell Studio will display the following dialog if this is required:



You can select from the following options:

- **Append (Default / Recommended)**

Leaves the event assignment as is, but also assigns the control set's event block via the script editor—or—*this allows the control to assign more than one script block to the event.*

- **Leave**

Leaves the event assignment as is.

- **Replace**

Replaces the existing assignment in the designer with the control set's new event block.

### Event Dialog

There are three options in the Control Sets' *Replace Event* dialog: **Append**, **Leave**, or **Replace**.

When you select **Append**, the form's event is assigned below the new event script block:

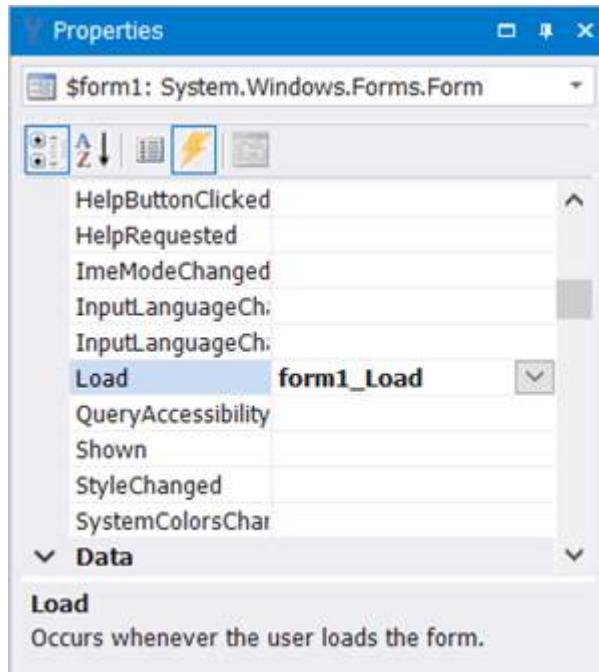
```
1
2 $form1_Load={
3     #TODO: Initialize Form Controls here
4
5 }
6
7 $buttonLoadProcess_Click={
8     #TODO: Place custom script here
9     $textbox2.Text = Get-Process | Out-String
10
11
12 $fadeIn_Load={
13     #Start the Timer to Fade In
14     $timerFadeIn.Start()
15     $form1.Opacity = 0
16 }
17 #Append the event to the form
18 $form1.add_Load($fadeIn_Load)
19
```

👉 This allows you to use multiple control sets that share the same events without them interfering with one another.

In the code shown below, the form *Load* event handler (\$form1\_Load) has been customized to write information to the debug pipeline:

```
1
2 $form1_Load={
3     #TODO: Initialize Form Controls here
4     Write-Debug "Form loaded at $(Get-Date) by $env:username"
5 }
6
```

We can see from the form properties that this handler has been connected to the form's *Load* event:

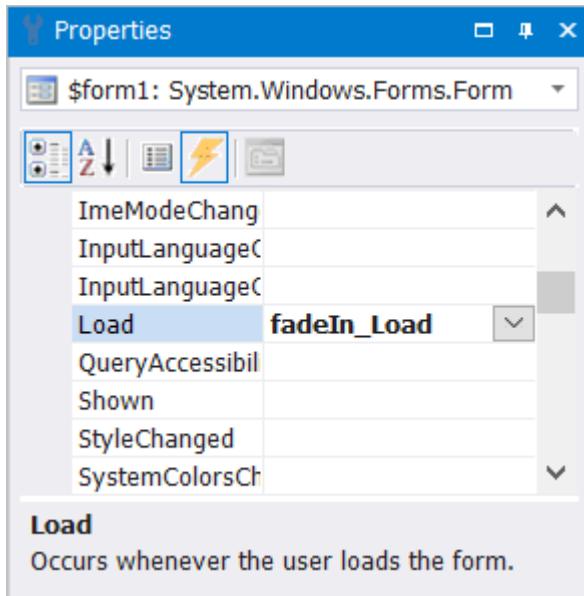


If we now add the 'Fade in Effect' control set and choose **Replace**, the following changes occur:

1. New code is added, without affecting any existing code:

```
1
2 $form1_Load={
3     #TODO: Initialize Form Controls here
4     Write-Debug "Form loaded at $(Get-Date) by $env:username"
5 }
6
7
8 $fadeIn_Load={
9     #Start the Timer to Fade In
10    $timerFadeIn.Start()
11    $form1.Opacity = 0
12 }
13
14 $timerFadeIn_Tick={
15     #Can you see me now?
16     if($form1.Opacity -lt 1)
17     {
18         $form1.Opacity += 0.1
19
20         if($form1.Opacity -ge 1)
21         {
22             #Stop the timer once we are 100% visible
23             $timerFadeIn.Stop()
24         }
25     }
26 }
27 }
```

2. The form's *Load* event is now connected to the new code:



- ❗ Now the Write-Debug statement is not executed because the original load event will not be called.

If we had selected **Leave** instead of **Replace** in the example above, the control set code would have been added without connecting the form's *Load* event to the new event handler, thus leaving it to us to connect things as we want.

- 👍 A simple code addition will allow us to call the original event, or a new event. For example, adding the following line will execute the code stored in the variable \$form1\_fadeInLoad:

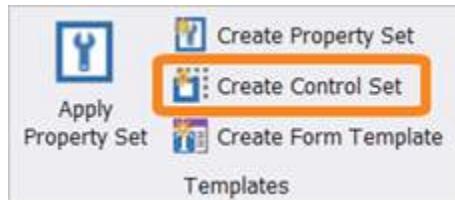
&\$form1\_fadeInLoad

```
1
2 $form1_Load={
3     #TODO: Initialize Form Controls here
4     Write-Debug "Form loaded at $(Get-Date) by $env:username"
5     &$form1_fadeInLoad
6 }
7
8 $fadeIn_Load={
9     #Start the Timer to Fade In
10    $timerFadeIn.Start()
11    $form1.Opacity = 0
12 }
13
```

## Creating Control Sets

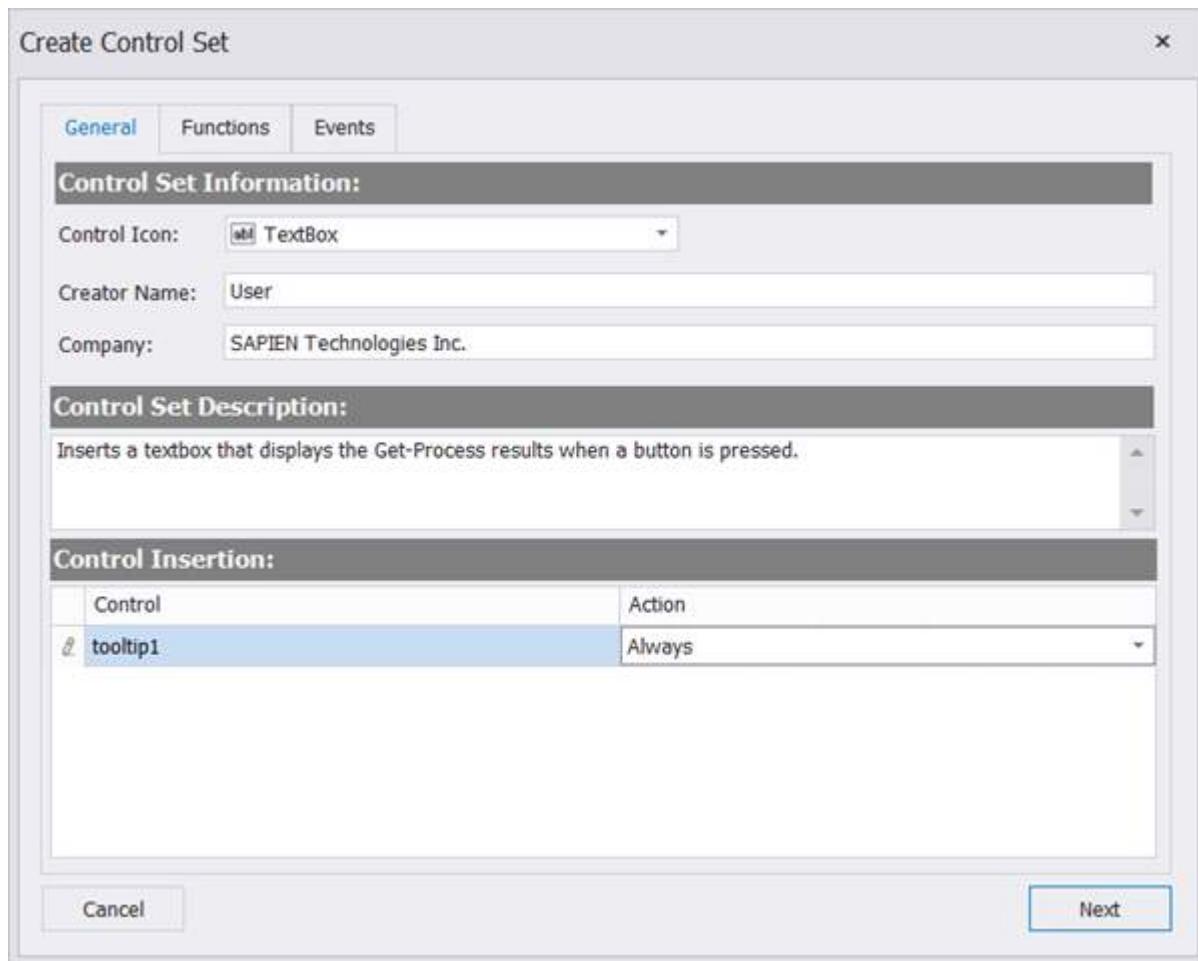
Creating your own control sets is another way to build reusable components for your scripting tasks.

To create a control set, add controls to a form, configure their properties, and write all of the code required to make your new control set function correctly. Then select all of the controls that are to be included in the control set and click **Create Control Set** on the Designer ribbon:



Or you can right-click on the Designer and select **Create Control Set (Ctrl+T)**.

The Create Control Set dialog allows you to configure the new control set:



Configure the options in the Create Control Set dialog (**General** tab) and then select **Next**:

- **Control Icon**

Choose the icon to associate with the control when it is displayed in the [Toolbox panel](#)  [250].

- **Creator Name**

Your name. (The value for this field is taken from **Home > Options > General > Username**).

- **Company**

Your company details. (The value for this field is taken from **Home > Options > General > Organization**).

- **Control Set Description**

Provide a description for the control set. *This field is mandatory.*

- **Control Insertion**

If you have included any non-visual controls you can use the **Control Insertion** section to define what happens when your custom control is added to a form. You have three choices as described below.

- a. **Always**

If you add your new custom control to a form that already has a control called timer1, PowerShell Studio will rename the new timer and modify the code in your custom control to use the new name.

- b. **Use Existing Type**

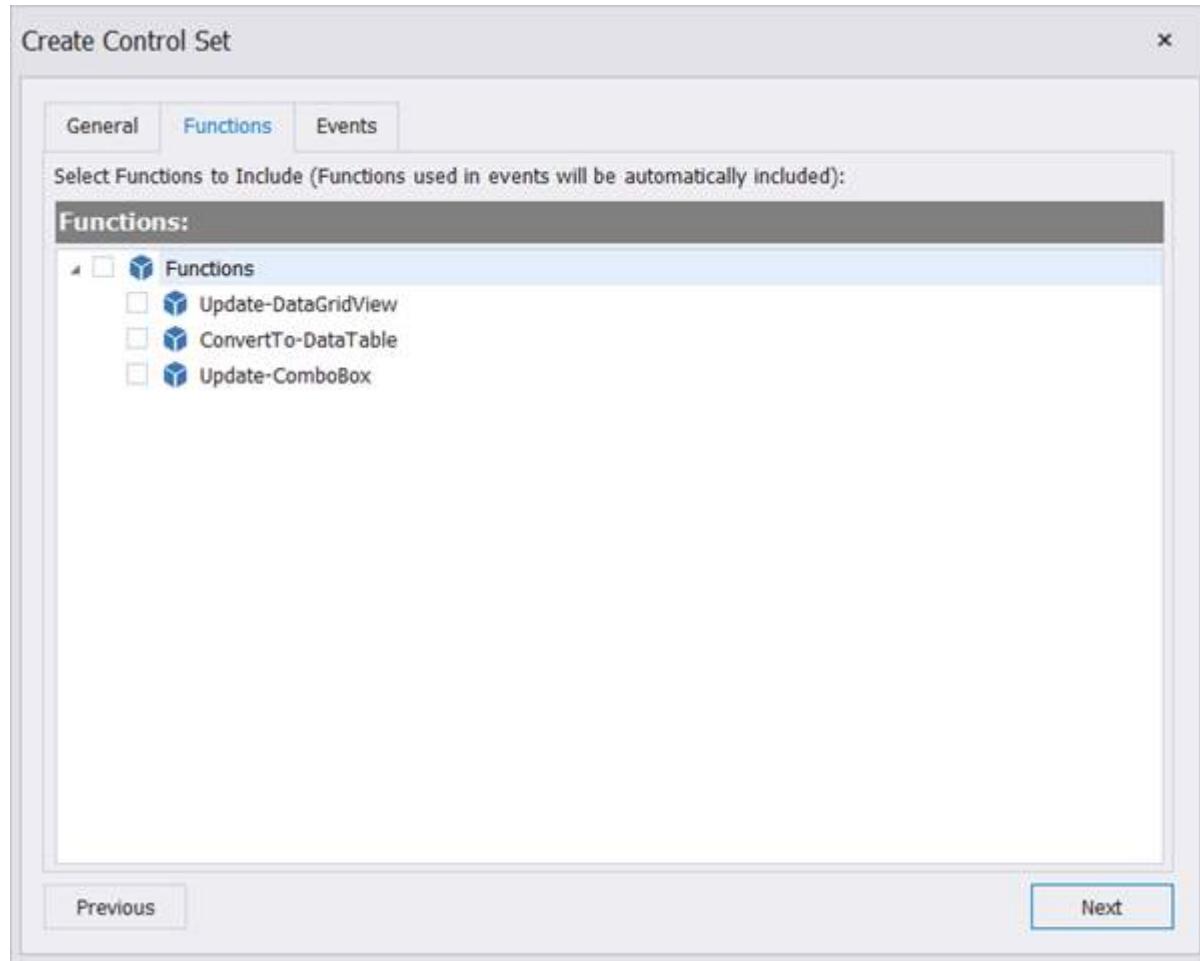
If the destination form already has a control of the same type (Timer in the example) then PowerShell Studio will not add a new control but rather modify the code to refer to the existing control.

- c. **Use Existing Type (Match Name)**

If the destination form already has a control of the same type (Timer in the example) and the name matches the name in your custom control, PowerShell Studio will behave as in b (i.e., use the existing control).

If the names do not match, then PowerShell Studio will add everything in your custom control to the form (i.e., it will behave as in a).

On the **Functions** tab you can choose to include any functions from your code in the template. PowerShell Studio will automatically include functions that are bound to events in the control:

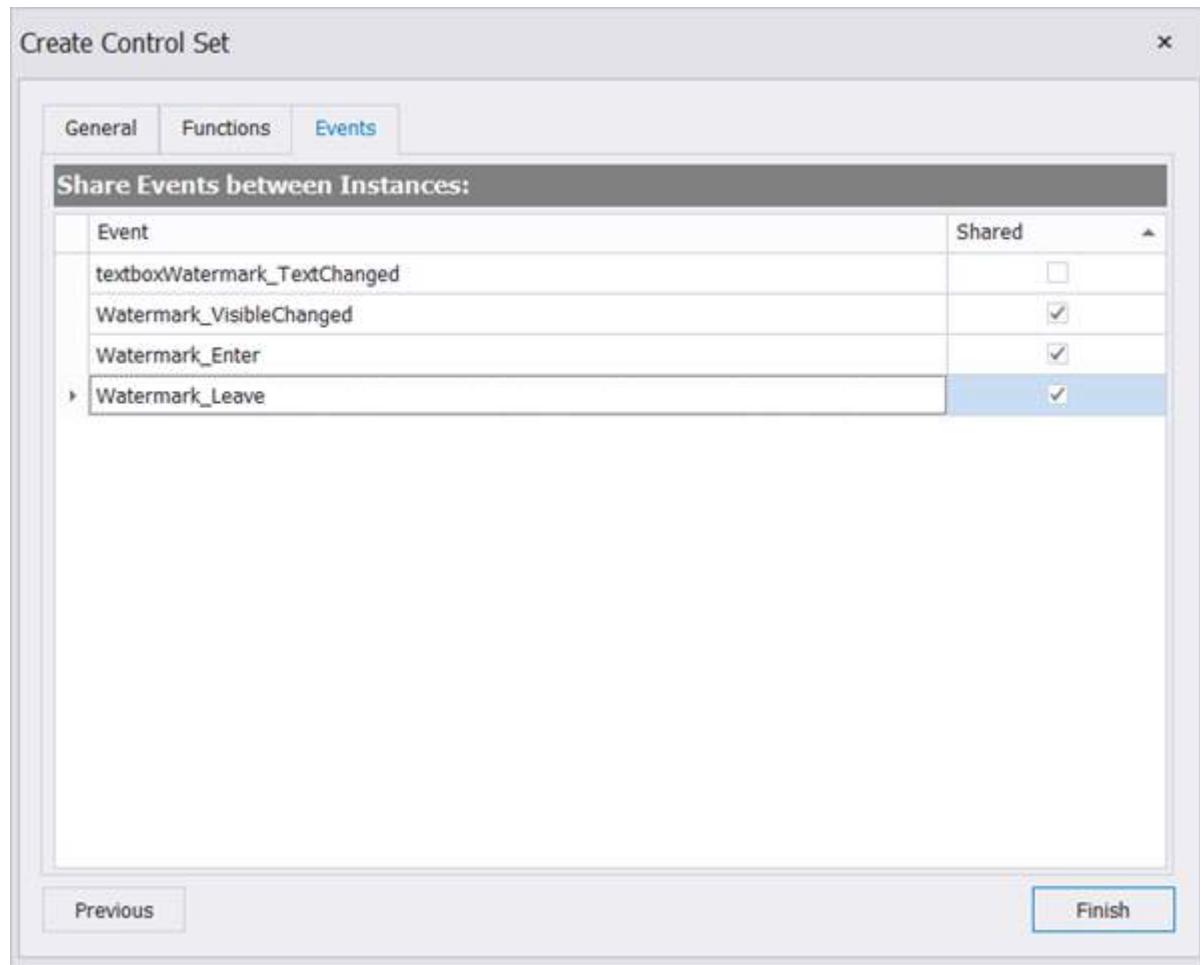


Click **Next** to go to the Events tab where you can mark an event as shared.

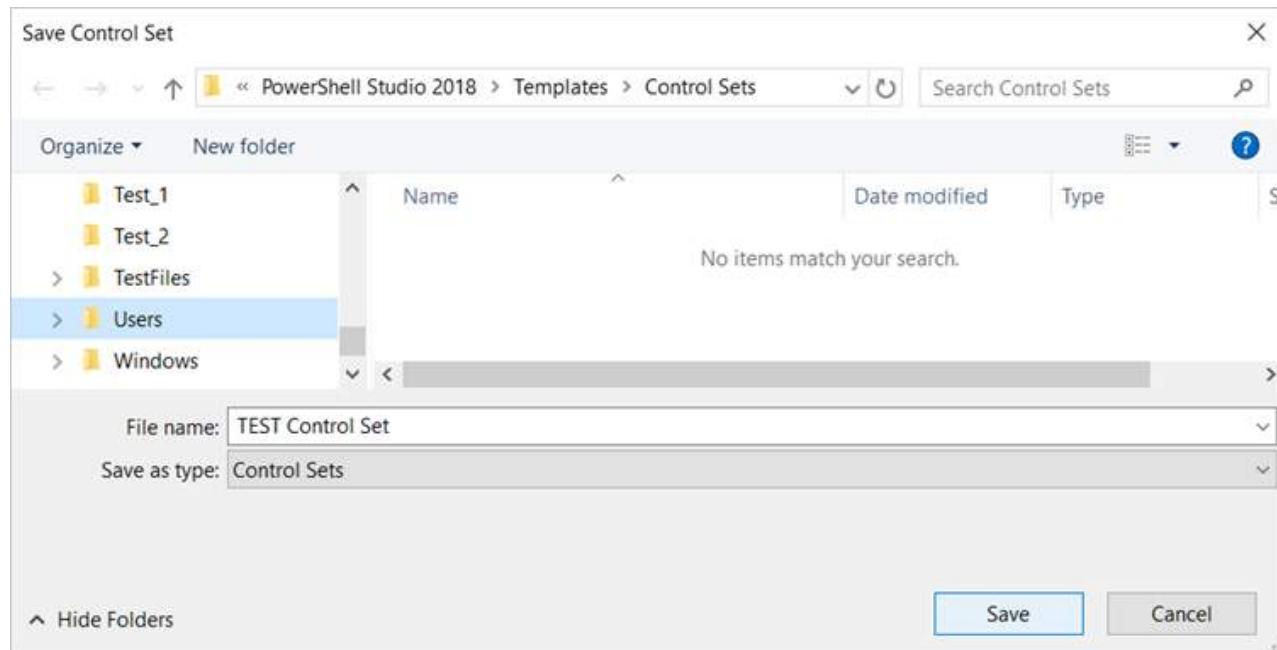
When you check the **Shared** checkbox next to an event, it will tell PowerShell Studio to share the event over multiple instances of the control set. It does this by first checking if the event already exists:

- *If the event does not exist*, then it will insert the event.
- *If the event does exist*, it will simply use the existing script block whenever a control triggers the event.

The *TextBox-Watermark* control set in PowerShell Studio serves as a good example for event sharing. The script blocks to display the watermark are identical for all instances of the control set, and therefore it doesn't make sense to create a new instance of the same event block every time the control set is inserted.



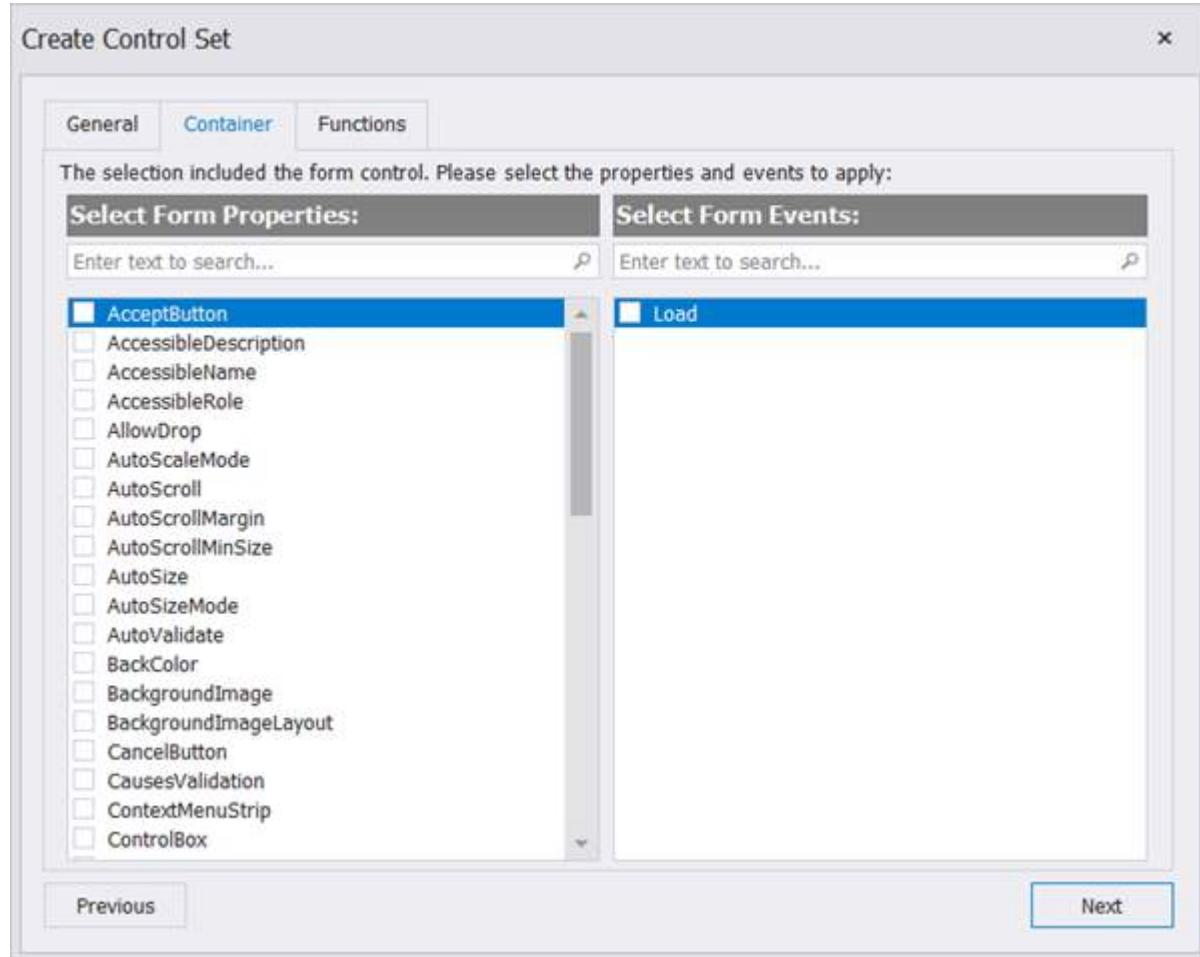
Click Finish, enter a file name and click Save:



- i** The default location for custom control sets is %AppData%\SAPIEN\PowerShell Studio <year>\Templates\Control Sets.

The control set is added to the Control Sets tab in the [Toolbox panel](#).

If you include an entire form in a control set, PowerShell Studio will add the Container tab to the Create Control Set dialog:



The Container tab helps you to specify which form property values and event handlers should be included in the control set.

- 👍** The next time you need to use your new control set, simply drag it from the [Toolbox panel](#) onto a form.

## 8 Panels

PowerShell Studio has dockable window panels that are used by dedicated features or to display output. This section provides an overview of the available panels and their content.

### PowerShell Studio Panels

---

Panels available in PowerShell Studio:

<i>Panel</i>	<i>Keyboard Shortcut</i>	<i>Description</i>
<a href="#">Call Stack</a> <sup>[206]</sup>	<i>Ctrl + Alt + P, K</i>	Displays the function or procedure calls that are currently on the stack. Used in debugging.
<a href="#">Console</a> <sup>[206]</sup>	<i>Ctrl + Alt + P, C</i>	Hosts PowerShell and other embedded consoles in a separate process.
<a href="#">Debug Console</a> <sup>[208]</sup>	<i>Ctrl + Alt + P, D</i>	A customizable command line console (PowerShell, PSCore, Bash, etc.) that allows you to interact with a debug session when at a breakpoint.
<a href="#">Find Results</a> <sup>[209]</sup>	<i>Ctrl + Alt + P, R</i>	Displays <a href="#">Find in Files</a> <sup>[62]</sup> and <a href="#">Find All References</a> <sup>[66]</sup> search results.
<a href="#">Function Explorer</a> <sup>[210]</sup>	<i>Ctrl + Alt + P, F</i>	Lists all functions, events, workflows, and configurations referenced in the current file. When working in a project, functions defined in other project files are also displayed.
<a href="#">Help</a> <sup>[212]</sup>	<i>Ctrl + Alt + P, H</i>	Displays Windows PowerShell command line help and WMI Help (F1).
<a href="#">Object Browser</a> <sup>[214]</sup>	<i>Ctrl + Alt + P, B</i>	Displays Windows PowerShell modules and commands, .NET Framework types, WMI objects, and database objects.
<a href="#">Output</a> <sup>[221]</sup>	<i>Ctrl + Alt + P, O</i>	Displays all script output including general application messages, build information, errors, debug, verbose, and trace-point output.
<a href="#">Performance</a> <sup>[223]</sup>	<i>Ctrl + Alt + P, M</i>	Displays the CPU and memory usage of your PowerShell scripts.
<a href="#">Project</a> <sup>[225]</sup>	<i>Ctrl + Alt + P, J</i>	Central location for managing projects, including the project's files and folders.
<a href="#">Properties</a> <sup>[240]</sup>	<i>Ctrl + Alt + P, P</i>	View and edit the control properties when working in the GUI Designer. Edit project settings and project file settings when working in the PowerShell Studio - Help Manual

👉 To quickly access a panel, execute the associated chorded keyboard shortcut. Simply press ***Ctrl+Alt+P***, release, then press the corresponding character key of the chord.

## 8.1 Call Stack Panel

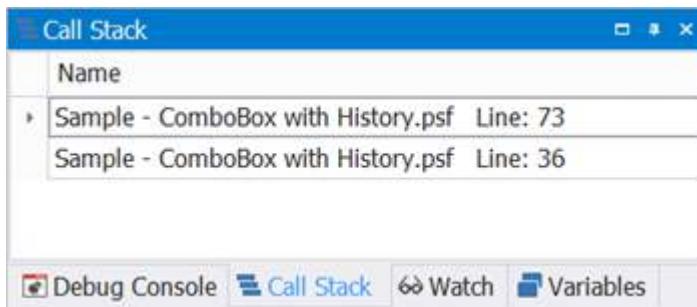
The Call Stack panel is used during debugging to display the function or procedure calls that are currently on the stack.

### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***K***.

### Call Stack Panel Overview

The Call Stack panel displays the execution path through your script to the current breakpoint when debugging. Each line, except the first, represents a point where your script called a function. The first line is the location of the current breakpoint. Double-clicking on a line in this window will take you to that distinct line in the code editor:



## 8.2 Console Panel

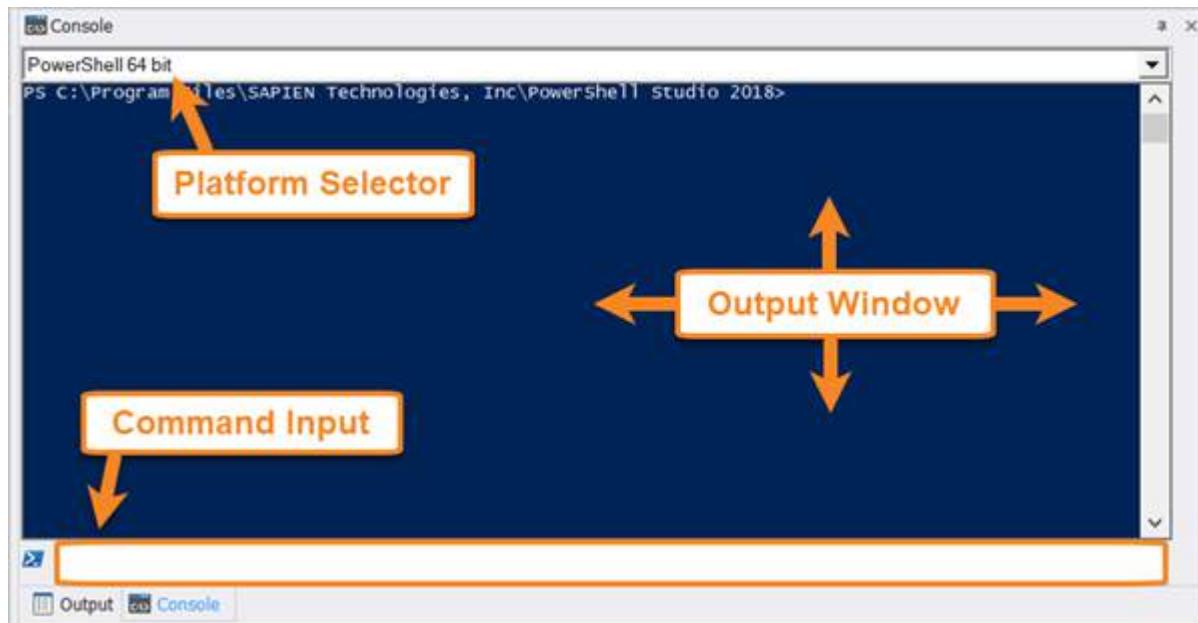
The Console panel hosts PowerShell and other embedded consoles in a separate process.

### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***C***.

### Console Panel Overview

You can interact with your script in the Console panel, which has three component parts:



- **Platform Selector**

Select from the list to activate a preconfigured console.

- **Output Window**

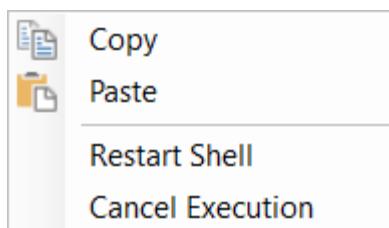
The Console output window displays the results of any command you send to the shell.

- **Command Input**

Type commands here and press < Enter > to send them to the shell. By default this option is disabled. To enable Command Input go to the Home tab > Options dialog > Console tab and select **Enable enhanced console input line**.

## Console Panel - Context Menu Options

Right-click in the Console window to display the following options:



- **Copy**

Copy text highlighted in the Console window to the clipboard.

- **Paste**

Copy the contents of the clipboard into the console.

- **Restart Shell**

Restart the shell. This will erase all work done in the current shell.

- **Cancel Execution**

Cancel the script / command that is executing in the console.

### 8.3 Debug Console

The Debug Console is a customizable command line console (PowerShell, PSCore, Bash, etc.) that allows you to interact with a debug session when at a breakpoint.

#### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***D***.

### Debug Console Overview

With the Debug Console, you can interact with the RunSpace while at a breakpoint. The console allows you to run commands or alter values in order to make debugging scripts easier. You can also simply experiment with what-if scenarios.

The screenshot shows the PowerShell Studio Debug Console window. At the top, there is a command prompt with the text: >> \$SPS = Get-Process -Name "PowerShell Studio". Below the prompt is a table displaying process information:

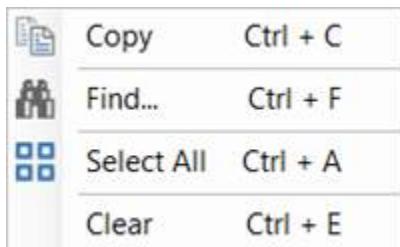
Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
1415	114	215236	315828	41.84	17576	13	PowerShell Studio

At the bottom of the window, there is a command history bar containing the same command (\$SPS = Get-Process -Name "PowerShell Studio"). Below the command history bar are tabs for "Debug Console", "Call Stack", "Watch", and "Variables".

👉 Arrow up and down to scroll through the history of commands in the active session.

### Debug Console - Context Menu Options

Right-click in the Debug Console window to display the following options:



- **Copy (Ctrl+C)**  
Copy highlighted text to the clipboard.
- **Find (Ctrl+F)**  
Search the text in the Debug Console panel.
- **Select All (Ctrl+A)**

Select all of the text in the Debug Console panel.

- **Clear (Ctrl+E)**

Clear the Debug Console panel.

## 8.4 Find Results Panel

The Find Results panel displays [Find in Files](#)<sup>62</sup> and [Find All References](#)<sup>66</sup> search results.

### Keyboard Shortcut

Press **Ctrl + Alt + P**, release, then press **R**.

## Find Results Panel Overview

The Find Results panel automatically displays results generated from queries performed using the [Find in Files](#)<sup>62</sup> dialog and the Script Editor's [Find All References](#)<sup>66</sup> option.

The results displayed include the associated file path, file name, and line number (#):

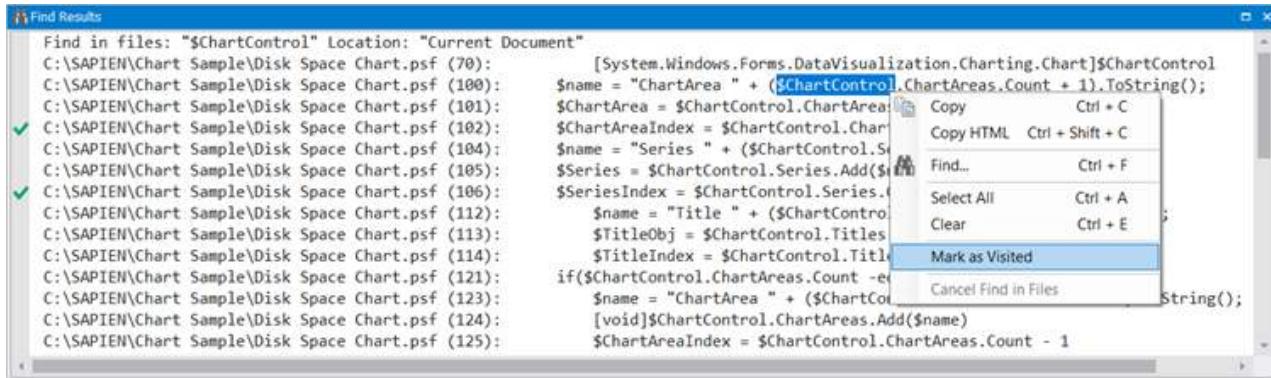
```
Find in files: "$ChartControl" Location: "Current Document"
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (70): [System.Windows.Forms.DataVisualization.Charting.Chart]$ChartControl
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (100): $name = "ChartArea " + ($ChartControl.ChartAreas.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (101): $ChartArea = $ChartControl.ChartAreas.Add($name)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (102): $ChartAreaIndex = $ChartControl.ChartAreas.Count - 1
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (104): $name = "Series " + ($ChartControl.Series.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (105): $Series = $ChartControl.Series.Add($name)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (106): $SeriesIndex = $ChartControl.Series.Count - 1
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (112): $name = "Title " + ($ChartControl.Titles.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (113): $TitleObj = $ChartControl.Titles.Add($name)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (114): $TitleIndex = $ChartControl.Titles.Count - 1
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (121): if($ChartControl.ChartAreas.Count -eq 0)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (123): $name = "ChartArea " + ($ChartControl.ChartAreas.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (124): [void]$ChartControl.ChartAreas.Add($name)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (125): $ChartAreaIndex = $ChartControl.ChartAreas.Count - 1
```

Double-click on a search result to view the referenced line in the script. Viewed results are distinguished by a green check mark on the left column:

```
Find in files: "$ChartControl" Location: "Current Document"
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (70): [System.Windows.Forms.DataVisualization.Charting.Chart]$ChartControl
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (100): $name = "ChartArea " + ($ChartControl.ChartAreas.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (101): $ChartArea = $ChartControl.ChartAreas.Add($name)
✓ C:\SAPIEN\Chart Sample\Disk Space Chart.psf (102): $ChartAreaIndex = $ChartControl.ChartAreas.Count - 1
✓ C:\SAPIEN\Chart Sample\Disk Space Chart.psf (104): $name = "Series " + ($ChartControl.Series.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (105): $Series = $ChartControl.Series.Add($name)
✓ C:\SAPIEN\Chart Sample\Disk Space Chart.psf (106): $SeriesIndex = $ChartControl.Series.Count - 1
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (112): $name = "Title " + ($ChartControl.Titles.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (113): $TitleObj = $ChartControl.Titles.Add($name)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (114): $TitleIndex = $ChartControl.Titles.Count - 1
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (121): if($ChartControl.ChartAreas.Count -eq 0)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (123): $name = "ChartArea " + ($ChartControl.ChartAreas.Count + 1).ToString();
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (124): [void]$ChartControl.ChartAreas.Add($name)
C:\SAPIEN\Chart Sample\Disk Space Chart.psf (125): $ChartAreaIndex = $ChartControl.ChartAreas.Count - 1
```

The check mark indicator helps you keep track of all the locations you visited.

To manually mark a result as *visited* or *unvisited*, right-click on a result and select **Mark as Visited** or **Mark as Unvisited**:



- i** If you are working with a project, references located in the other project files will also display in the Find Results panel.

## 8.5 Function Explorer Panel

The Function Explorer lists all functions, events, workflows, and configurations referenced in the current file. When working in a project, functions defined in other project files are also displayed.

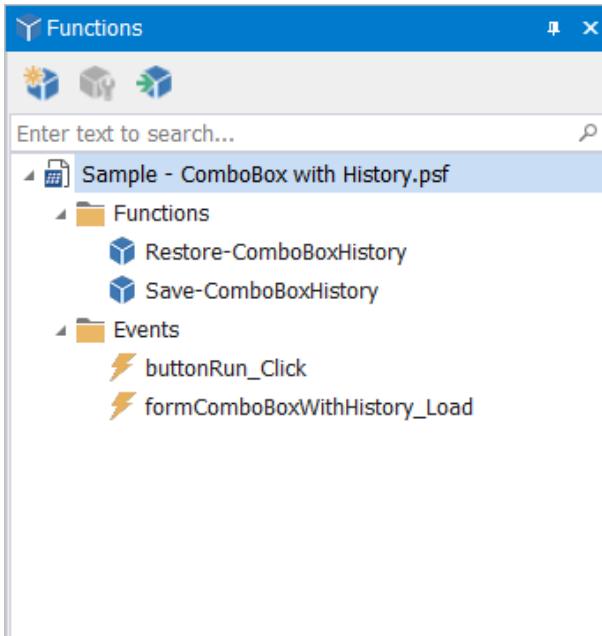
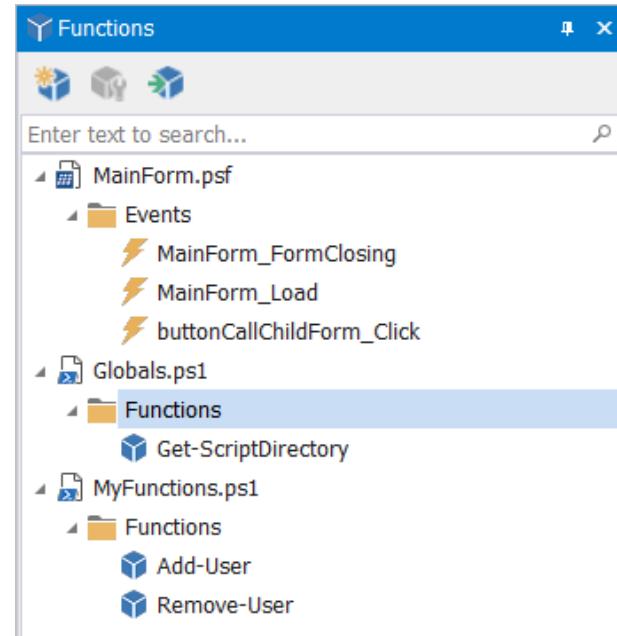
### Keyboard Shortcut

Press **Ctrl + Alt + P**, release, then press **F**.

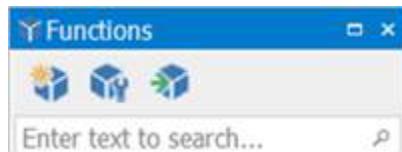
## Functions Panel Overview

The Functions panel displays all of the functions, events, and workflows in the current context.

When you are working on a single script, the functions and workflows within that script are displayed. When working in a project, all available functions in the project are displayed:

**Single Script****Project File****Functions Panel - Buttons and Search**

There are three buttons and a search box at the top of the Function Explorer panel:



From left to right:

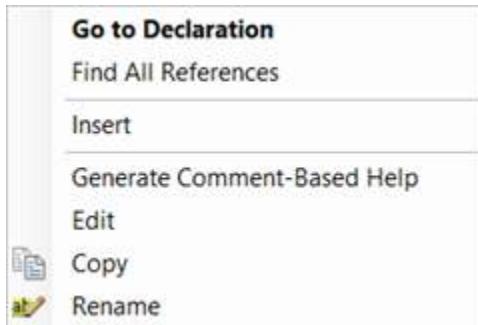
- **Insert Function**  
Opens the Function Builder where you can create a function.
- **Edit Function**  
Opens the highlighted function in the Function Builder.
- **Import Functions**  
Launches the Import Functions dialog along with an Explorer window for locating the file containing the functions to import.
- **Search**  
Search for a function, workflow, or event by typing the first few letters in the search box. As you type, PrimalSense™ will give you a list of possible completions.

Hover over a function to display the function name and details:



## Functions Panel - Context Menu Options

Right-click on a function to display the following options:



- **Go to Declaration**

Positions the caret in the relevant file at the source code for the selected function.

- **Find All References**

[Find all references](#) to this function in the script or project. Results are displayed in the [Find Results panel](#).

- **Insert**

Inserts a function call at the current caret position.

- **Generate Comment-Based Help**

Generates and inserts templated comment-based help for the selected function.

- **Edit**

Edit the function in the Function Builder.

- **Copy**

Copies the function definition to the clipboard.

- **Rename**

Renames the function and updates the references in the script or project.

## 8.6 Help Panel

The Help panel displays Windows PowerShell command line help and WMI Help (*F1*).

### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***H***.

## Help Panel - Buttons and Search

There are three buttons on the top-left of the Help panel:

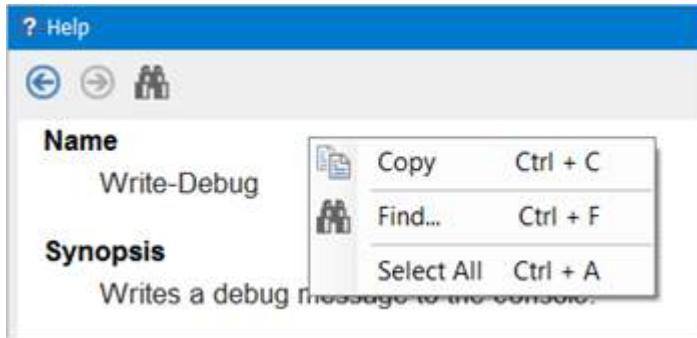


From left to right:

- **Go Back (Alt+Left)**  
Navigates backwards through the help files that have been loaded since PowerShell Studio was started.
- **Go Forward (Alt+Right)**  
Navigates forwards through the help files that have been loaded since PowerShell Studio was started.
- **Find (Ctrl+F)**  
Searches the text in the Help panel.

## Help Panel - Context Menu Options

Right-click in the Help panel to display the following options:



- **Copy (Ctrl+C)**  
Copies the highlighted text to the clipboard.
- **Find (Ctrl+F)**  
Searches the text in the Help panel.
- **Select All (Ctrl+A)**  
Selects all of the text in the Help panel.

## 8.7 Object Browser

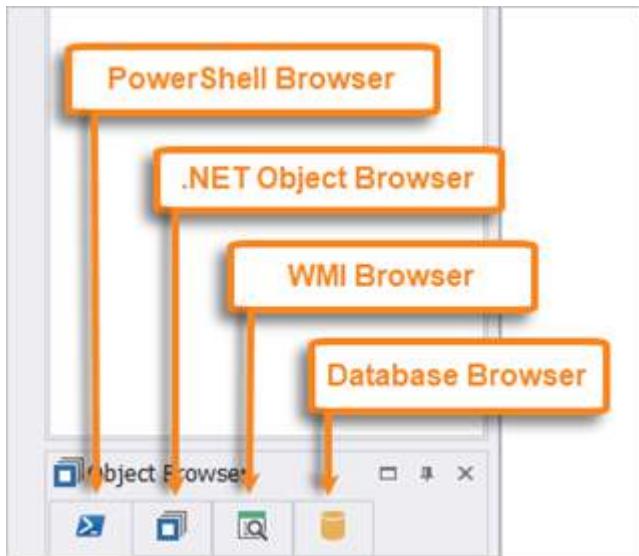
The Object Browser displays Windows PowerShell modules and commands, .NET Framework types, WMI objects, and database objects.

### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***B***.

## Object Browser Overview

The Object Browser is one of the most useful features of PowerShell Studio and contains the following browsers:



[PowerShell Browser](#)<sup>214</sup> For exploring PowerShell objects including cmdlets, aliases, modules, functions, and About Help topics.

[.NET Object Browser](#)<sup>216</sup> For exploring .NET types.

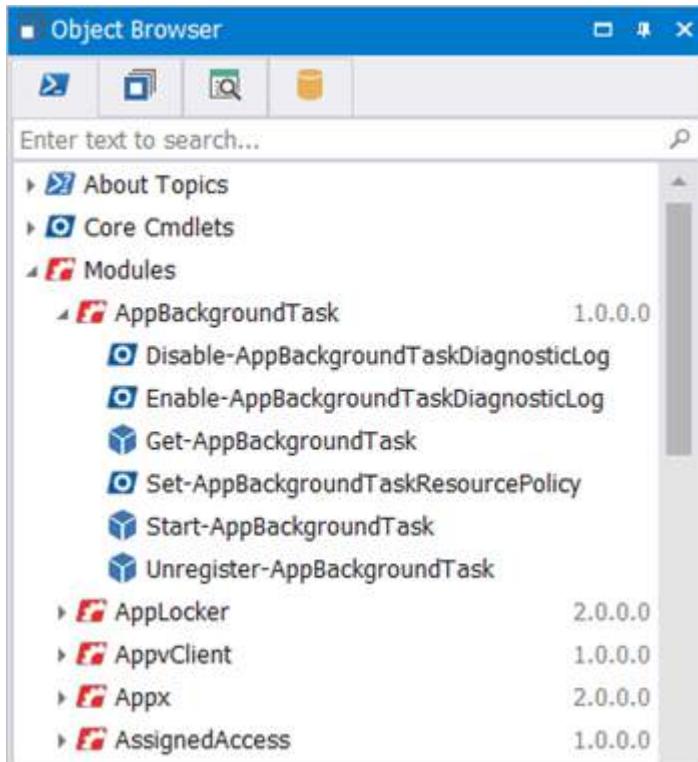
[WMI Browser](#)<sup>217</sup> For exploring the WMI database on your computer.

[Database Browser](#)<sup>218</sup> For exploring databases.

Each browser provides the same basic functionality—the ability to explore a collection of objects. Each browser also provides custom capabilities to help you integrate objects into your code.

### PowerShell Browser

The PowerShell browser displays PowerShell cmdlets, aliases, modules, and About Help topics:

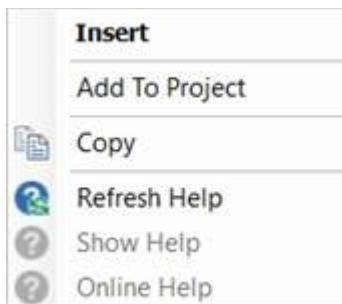


Search for a PowerShell cmdlet, alias, module, or About Help topic by typing the first few letters in the search box. As you type, PrimalSense™ will give you a list of possible completions:



## PowerShell Object - Context Menu Options

Right-click on a PowerShell object to display options available for the object:



- **Insert**

Inserts the selected command into a script file. If you insert a command from a module, then PowerShell Studio will also add the appropriate Import-Module command into your code.

- **Add To Project**

Inserts an import statement in the primary file of the project (*Startup.pss* or *Module.psm1*).

- **Copy**

Copies the object name to the clipboard.

- **Refresh Help**

Refresh Help will rebuild the cache help for the selected module. If no help is found, it will trigger this command automatically.

- **Show Help**

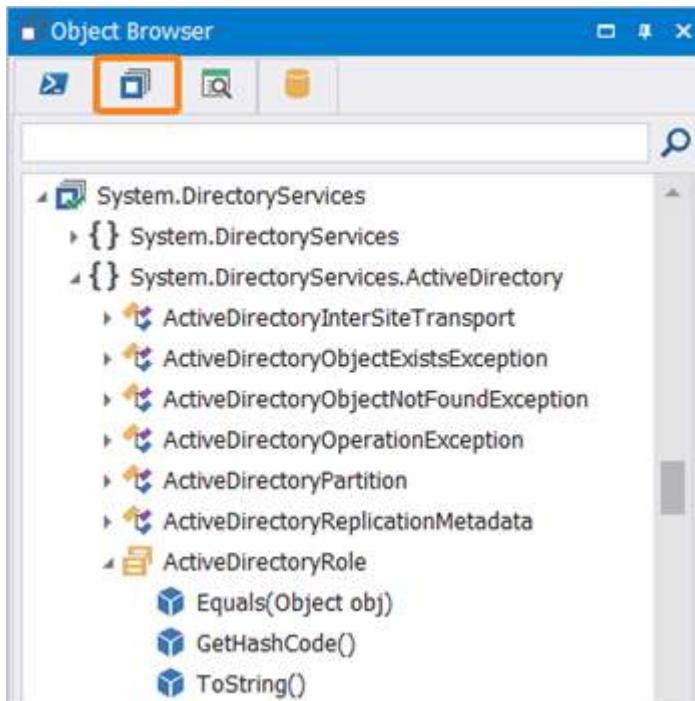
Displays PowerShell help about the object in the Help panel.

- **Online Help**

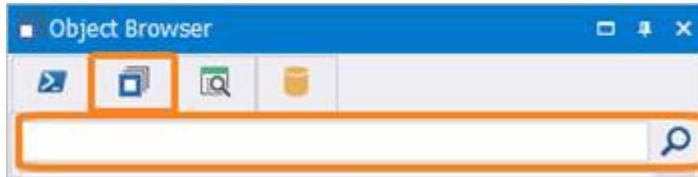
Displays help from the Microsoft web site in a browser window.

## .NET Object Browser

This browser displays the .NET assemblies available on your computer. Each assembly can be opened to reveal the namespaces and types contained within. Individual types can be opened to reveal their contents (properties, methods etc.):

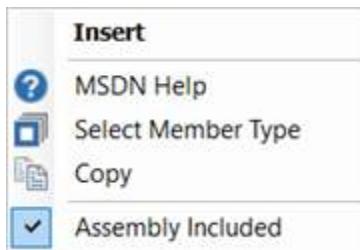


Search for .NET assemblies by typing the first few letters in the search box, then press < Enter > to see the first result. Continue pressing < Enter > to cycle through the search results:



## .NET Object - Context Menu Options

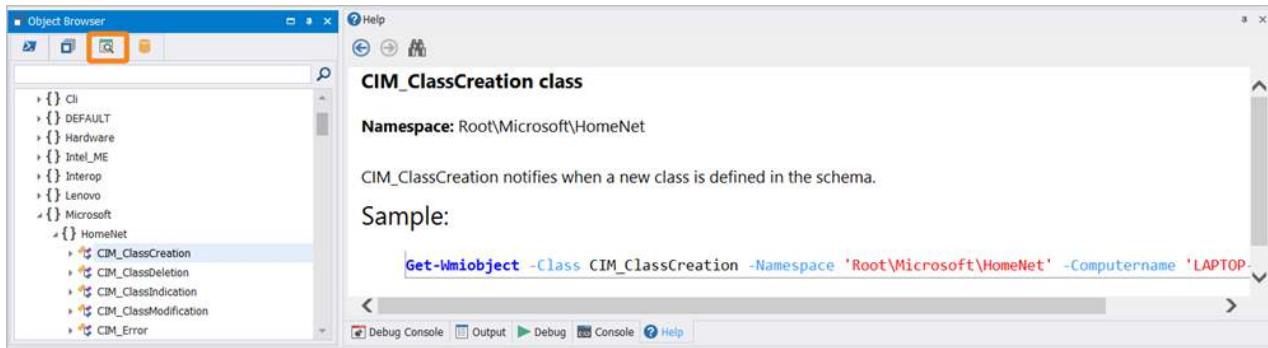
Right-click on a node to display the following options:



- **Insert**  
Inserts the name of the current node into your script.
- **MSDN Help**  
Accesses the MSDN help website for the selected type, method, property etc.
- **Select Member Type**  
Selects the current item's type, such as a Property's or Method's return type, in the .NET Object Browser.
- **Copy**  
Copies the current node name onto the clipboard.
- **Assembly Included**  
Indicates whether the assembly is loaded into the current document. You can check or uncheck the assembly in order to add or remove the assembly.

## WMI Browser

The WMI browser displays namespaces and objects from the WMI database on your computer. As you click on nodes in this panel, PowerShell Studio will display information and code examples about the node in the Help panel:

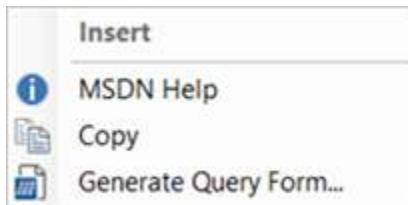


Look for namespaces or WMI objects by typing in the search box, then press < Enter > to see the first result. Continue pressing < Enter > to cycle through the search results:



## WMI Object - Context Menu Options

Right-click on a WMI object to display the following options:



- **Insert**

Inserts a Get-WMIObject command into a script to retrieve all instances of the selected WMI object.

- **MSDN Help**

Displays MSDN help for the selected WMI object.

- **Copy**

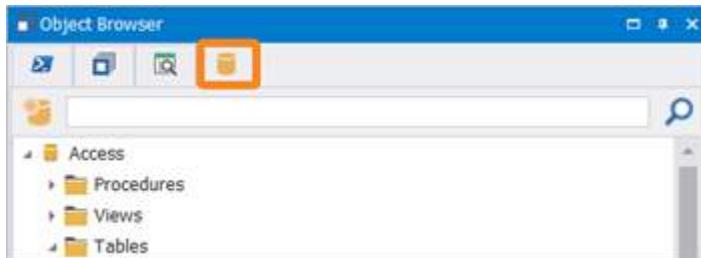
Copies the object name into the clipboard.

- **Generate Query Form...**

Generates a [form template](#) [171] that displays all instances of the selected WMI object in a grid.

## Database Browser

The database browser allows you to explore databases and generate code to read and display data:

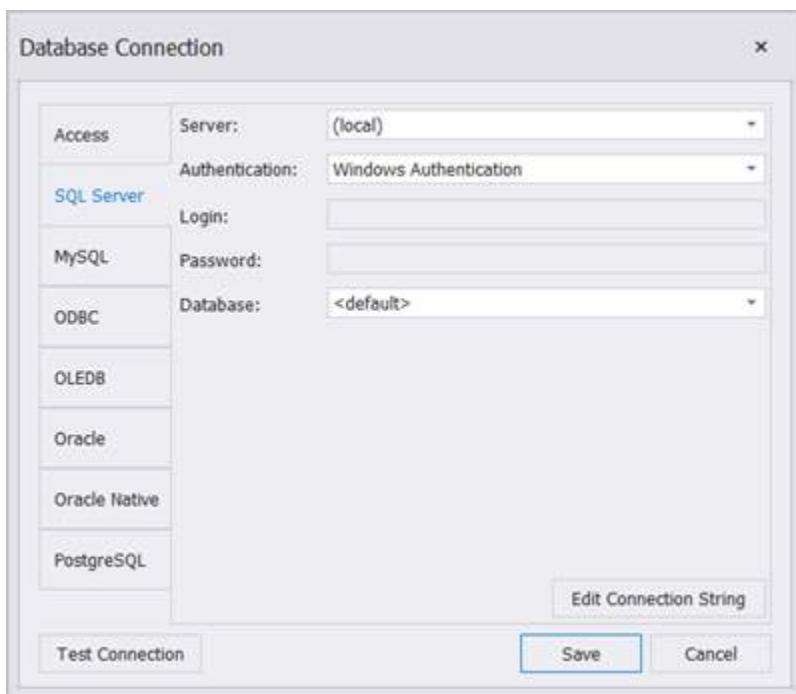
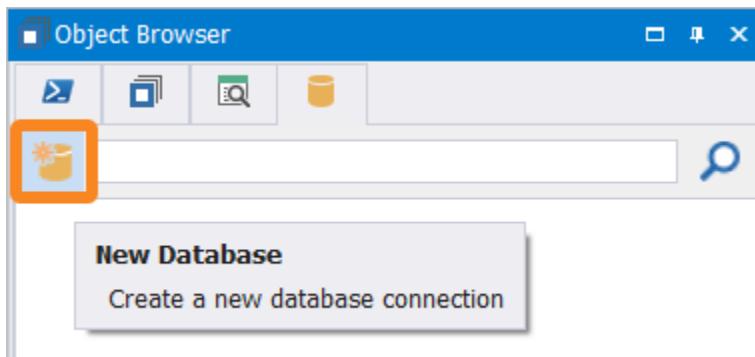


- i** The same database browser is shared with PrimalScript. Any connections created in PrimalScript will display in PowerShell Studio, and vice versa.

### How to create a database connection

The first step in using the database browser is to create a database connection.

Click on the create connection button ( ) to launch the connection dialog:



The code generators provide a default connection string that you will need to edit to reflect your environment. Once your database connection is configured, press **Save** to add a new node to the database browser. Click on the node to enumerate the contents of the database, which allows you to tunnel into the tables, views, stored procedures, etc.

Search for a database, table, or field by typing in the search box, then press < **Enter** > to see the first result. Continue pressing < **Enter** > to cycle through the search results:

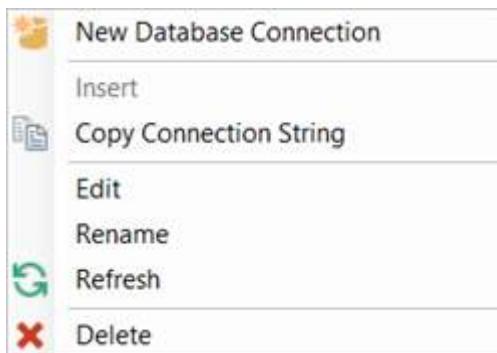


- The search function includes both active and cached database connections.

### Database Connection - Context Menu Options

---

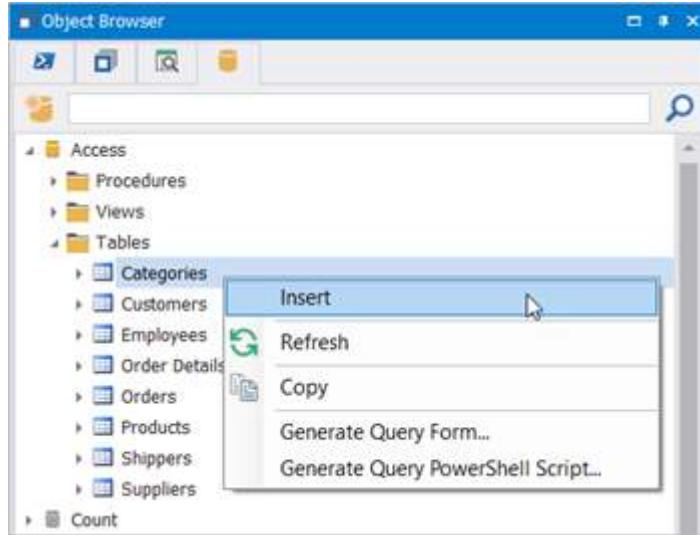
Right-click on a database connection to display the following options:



- **New Database Connection**  
Creates a new database connection.
- **Insert**  
Inserts the name into the Editor.
- **Copy Connection String**  
Copies the database connection string to the clipboard.
- **Edit**  
Edits the database connection.
- **Rename**  
Renames the database connection.
- **Refresh**  
Refreshes the Database Browser window.
- **Delete**  
Deletes the database connection.

## Database Objects - Context Menu Options

Right-click on a database object to display the following options:



- **Insert**

Generates a PowerShell function that uses ADO.Net to retrieve all records from the database and output the records to the PowerShell pipeline.

- **Refresh**

Refreshes the Database Browser window.

- **Copy**

Copies the name of the database object onto the clipboard.

- **Generate Query Form...**

Creates a PowerShell form based on available templates, and ADO.Net code, to retrieve database records into a .Net DataSet object and display the results in a grid.

- **Generate Query PowerShell Script...**

Creates a PowerShell script that uses ADO.Net to retrieve all records from the database and output the records to the PowerShell pipeline.

- i** Dragging and dropping a database object to the Script Editor will insert a function, whereas dragging and dropping a database object to the Designer will insert a grid with a function call.

## 8.8 Output Panel

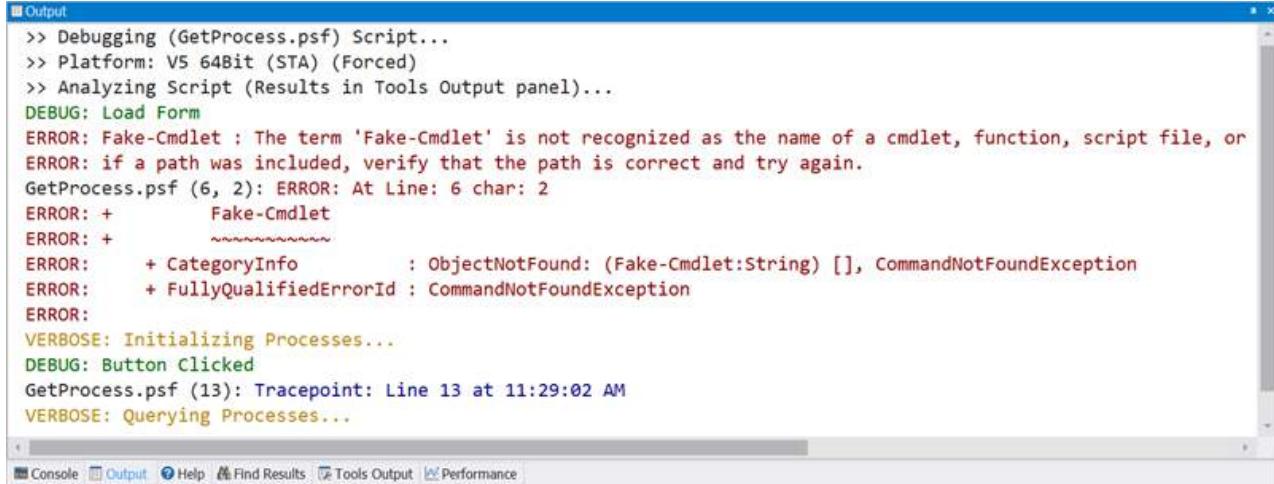
The Output panel displays all script output including general application messages, build information, errors, debug, verbose, and tracepoint output.

### Keyboard Shortcut

Press **Ctrl + Alt + P**, release, then press **O**.

## Output Panel Overview

The Output panel displays messages written to the debug stream (e.g., from the Write-Debug cmdlet), records messages created by tracepoints in your code, and displays verbose output:



The screenshot shows the Output panel in PowerShell Studio. It contains the following text:

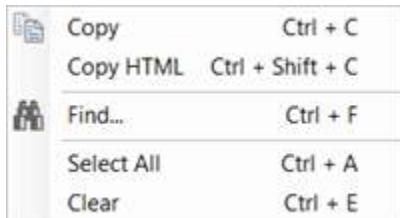
```
>> Debugging (GetProcess.psf) Script...
>> Platform: VS 64Bit (STA) (Forced)
>> Analyzing Script (Results in Tools Output panel)...
DEBUG: Load Form
ERROR: Fake-Cmdlet : The term 'Fake-Cmdlet' is not recognized as the name of a cmdlet, function, script file, or
ERROR: if a path was included, verify that the path is correct and try again.
GetProcess.psf (6, 2): ERROR: At Line: 6 char: 2
ERROR: +      Fake-Cmdlet
ERROR: + ~~~~~
ERROR:     + CategoryInfo          : ObjectNotFound: (Fake-Cmdlet:String) [], CommandNotFoundException
ERROR:     + FullyQualifiedErrorId : CommandNotFoundException
ERROR:
VERBOSE: Initializing Processes...
DEBUG: Button Clicked
GetProcess.psf (13): Tracepoint: Line 13 at 11:29:02 AM
VERBOSE: Querying Processes...
```

The bottom of the window shows tabs for Console, Output, Help, Find Results, Tools Output, and Performance.

👉 If the output contains the line number, you can double-click on an error message to go to that distinct line in the code editor.

## Output Panel - Context Menu Options

Right-click in the Output panel to display the following options:



- **Copy (Ctrl+C)**  
Copy highlighted text to the clipboard.
- **Copy HTML (Ctrl+Shift+C)**  
Copied highlight code, including color coding and formatting.
- **Find (Ctrl+F)**  
Search the text in the output panel.
- **Select All (Ctrl+A)**  
Select all of the text in the output panel.

- **Clear (Ctrl+E)**

Clear the output panel.

## 8.9 Performance Panel

The Performance panel displays the CPU and memory usage of your PowerShell scripts..

### Keyboard Shortcut

Press **Ctrl + Alt + P**, release, then press **M**.

### Performance Panel Overview

The Performance panel displays the CPU and memory usage of your scripts when you have performance monitoring enabled.

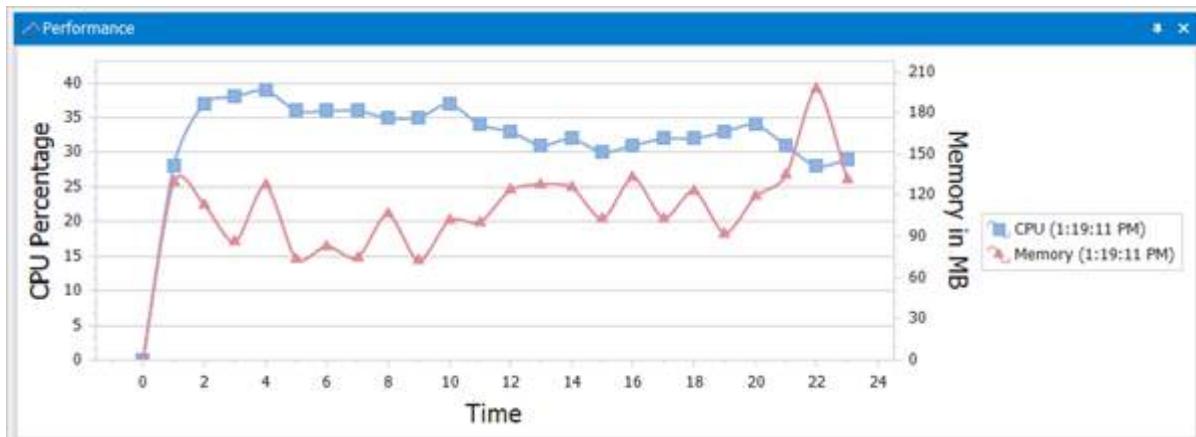
**i** The performance monitor does not record the statistics during debugging because the results would be skewed due to line breaking and variable querying.

### How to enable performance monitoring

To enable performance monitoring when running scripts, click the **Home** tab on the ribbon bar, then in the Run section select the **Monitor** checkbox:

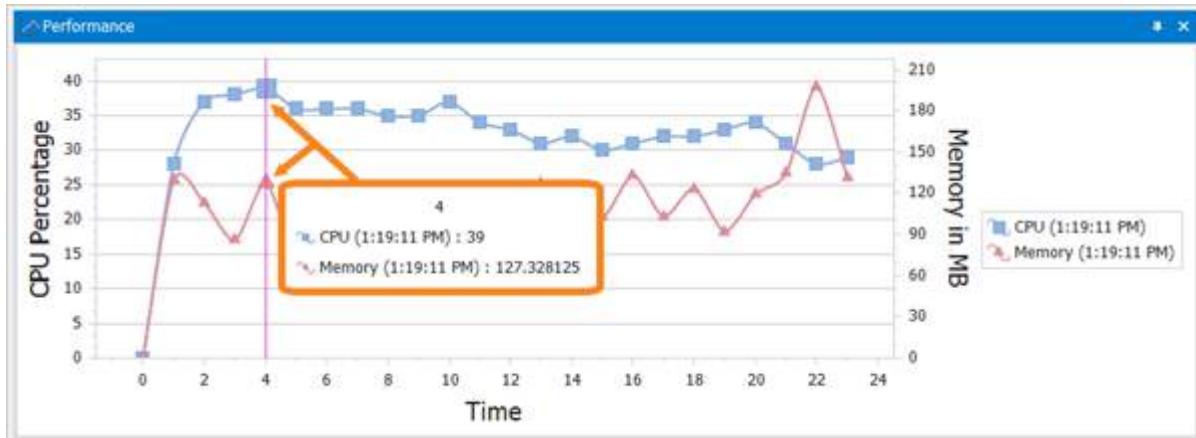


When performance monitoring is enabled, the CPU and memory usage of your scripts are displayed graphically in real-time during script execution:

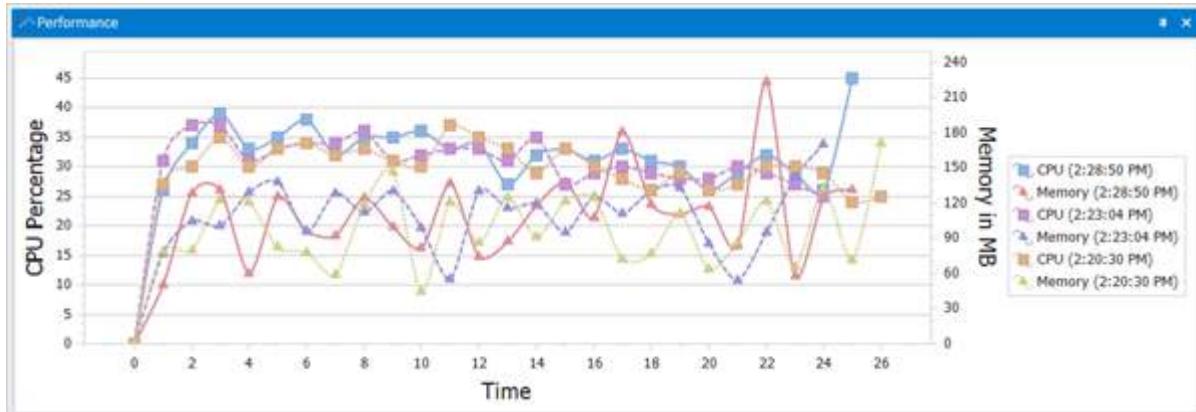


### How to view specific performance data

Scroll your pointer across the graph and hover over a data point to see the CPU and memory usage details:



The Performance panel will display the last three results of a script so that you can compare the performance differences of each run through:



### How to clear the performance data

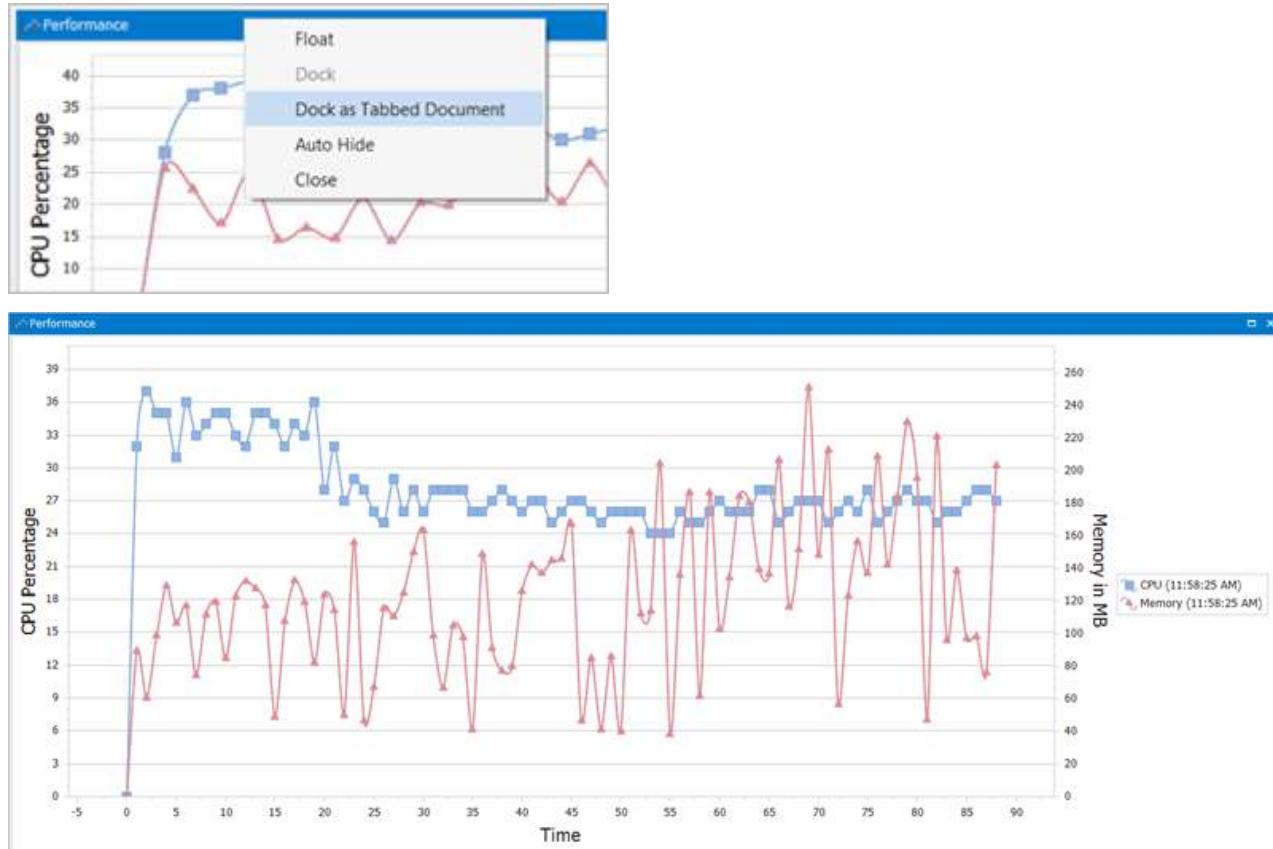
To clear the data from the Performance panel, right-click anywhere in the panel and select **Clear**:



### How to size the Performance panel

The graph in the docked Performance panel scales automatically as the script runs, but some details may not be visible if the docked window is sized at a small height. To see the performance data de-

tails in a broader view, depending on the current Performance panel location either right-click on the Performance panel title bar and select **Dock as Tabbed Document**, or **Float**; or drag the Performance panel tab away from the tabbed group and release to *float* the panel.



For information on docking and undocking panels, see [Working with Panels](#) [32]

- ➊ When the script ends the peak values that occurred during script execution are displayed in the Output panel:

```
Output
*** PowerShell Script finished. ***
>> Execution time: 00:00:23
>> Script Ended
>> Max. CPU: 39 % Max. Memory: 170.93 MB
```

## 8.10 Project Panel

The Project panel is a Central location for managing projects, including the project's files and folders.

## Keyboard Shortcut

Press **Ctrl + Alt + P**, release, then press **J**.

## Project Panel - Buttons and Search

There are four buttons and a search box at the top of the Project panel:

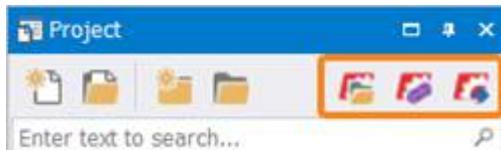


From left to right:

- **New File** launches the Add File dialog where you can select a file template and add a new file to the project.
- **Add Existing File** launches the file browser so that you can select a file to add to the project.
- **New Folder** allows you to add a new folder directly to the project.
- **Open Project Folder** launches File Explorer focused in the currently highlight folder.
- **Search** in a project by typing the first few letters in the search box. As you type, Prim-alSense™ will filter and display the files and folders containing your search criteria.

## Module Buttons

Three additional buttons are available at the top-right of the Project panel when you have a module project open:



From left to right:

- **Open Module Folder**  
Opens the exported module directory.
- **Clear Module Folder**  
Deletes the contents of the exported module directory.
- **Build Module**  
Builds and exports the module to the user's Windows PowerShell Modules directory: **C:\Users\<user>\Documents\WindowsPowerShell\Modules**

- i** The module manifest is automatically updated when you build the module project.

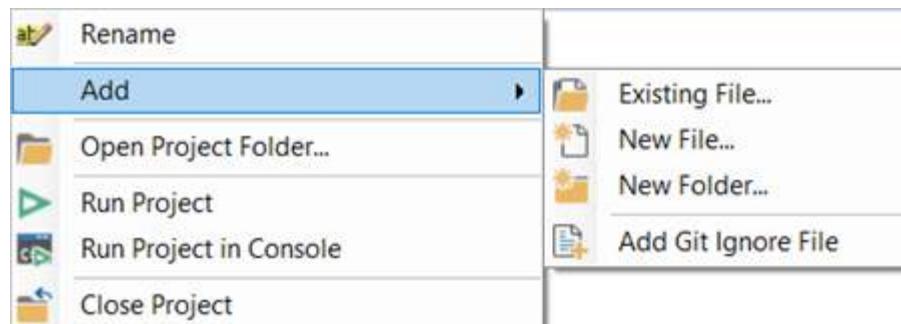
## Project Panel - Context Menu Options

The context menus in the Project panel provide useful options for building and working with projects.

- i** The context menu options available in the Project panel will vary depending on whether the item selected is a project, folder, or file, and the menu options will further vary by file type. If you select multiple files the context menu options will vary depending on the files selected.

### Project - Menu Options

Right-click on a project name to access the following options:



- **Rename**

Renames the project.

- **Add**

- **Existing File...**

Launches the file browser to select a file to add to the project.

- **New File...**

Launches the Add File dialog to add a new file to the project.

- **New Folder...**

Adds a new folder to the project.

- **Add Git Ignore File**

Creates a .gitignore file to filter out any temporary project files (for Git source control).

- **Open Project Folder...**

Opens the project directory in File Explorer.

- **Run Project**

Runs the project.

- **Run Project in Console**

Runs the project in the console window.

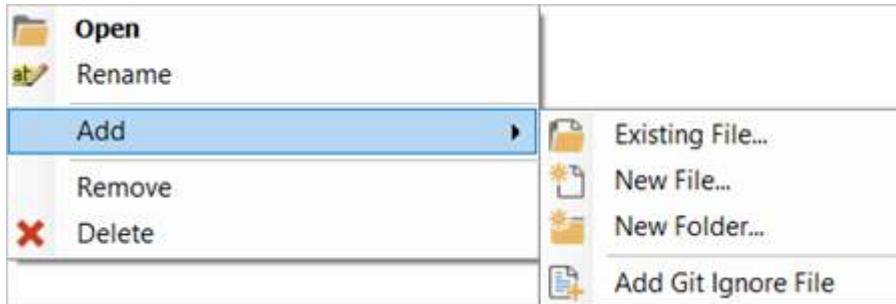
- **Close Project**

Closes the project.

## Project Folder - Menu Options

---

Right-click on a folder to access the following options:



- **Open**

Launches File Explorer focused in the currently highlight folder.

- **Rename**

Renames the folder and updates the references in the project.

- **Add**

- **Existing File...**

Launches the file browser to select a file to add to the project.

- **New File...**

Launches the Add File dialog to add a new file to the project.

- **New Folder...**

Adds a new folder under the highlighted location.

- **Add Git Ignore File**

Creates a .gitignore file to filter out any temporary project files (for Git source control).

- **Remove**

Removes the folder from the Project panel.

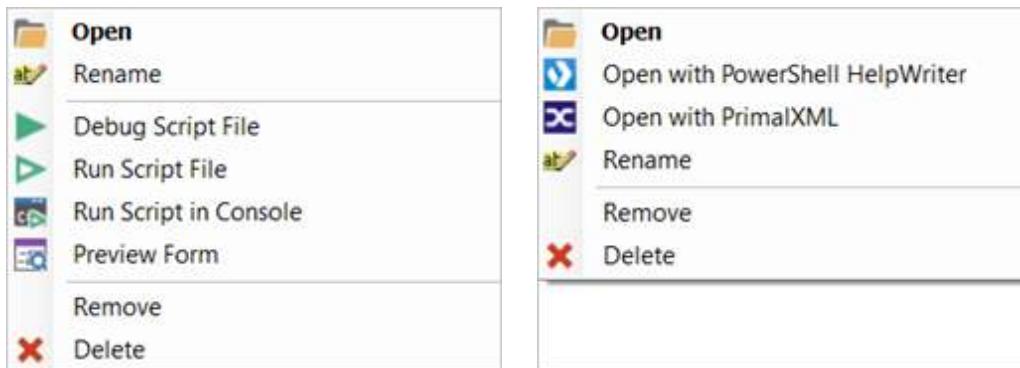
- **Delete**

Deletes the folder from the project directory and removes the folder from the Project panel.

## Project File - Menu Options

---

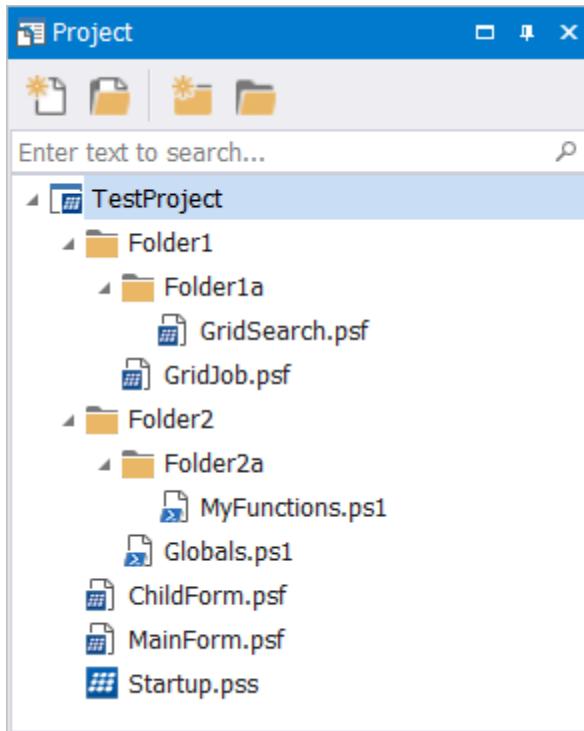
Right-click on a file to access context sensitive options based on the file type:



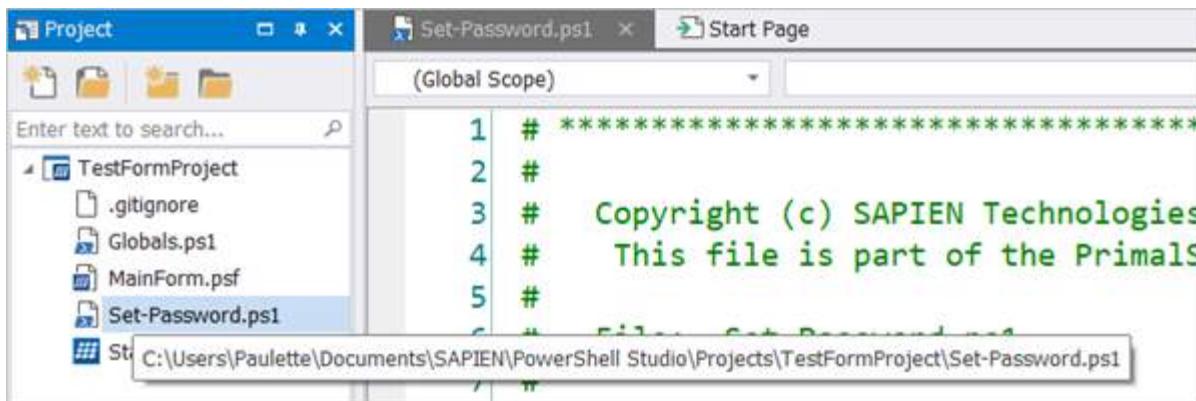
- **Open**  
Opens the file in the Editor or Designer.
- **Rename**  
Renames the file and updates the references in the project.
- **Debug Script File**  
Debugs the script.
- **Run Script File**  
Runs the script.
- **Run Script in Console**  
Runs the script in the console window.
- **Preview Form**  
Displays the form the way it will appear at run time without executing any code.
- **Remove**  
Removes the file from the Project panel.
- **Delete**  
Deletes the file from the project directory and removes the file from the Project panel.
- **Open with PowerShell HelpWriter**  
Opens the help file in PowerShell HelpWriter.
- **Open with PrimalXML**  
Opens the XML file in PrimalXML.

### 8.10.1 Project Files and Folders

Using folders within projects makes it easier to organize projects with large amounts of files. This topic shows you how to [add](#)<sup>[230]</sup>, [rename](#)<sup>[236]</sup>, [move](#)<sup>[237]</sup>, and [delete](#)<sup>[237]</sup> project files and folders.



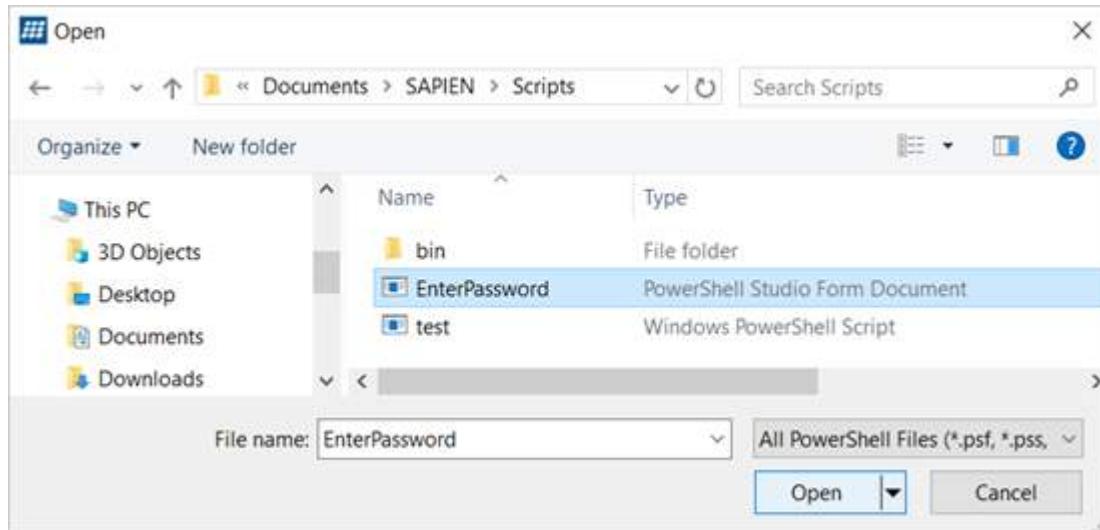
👉 Hover over a file to see the file path:



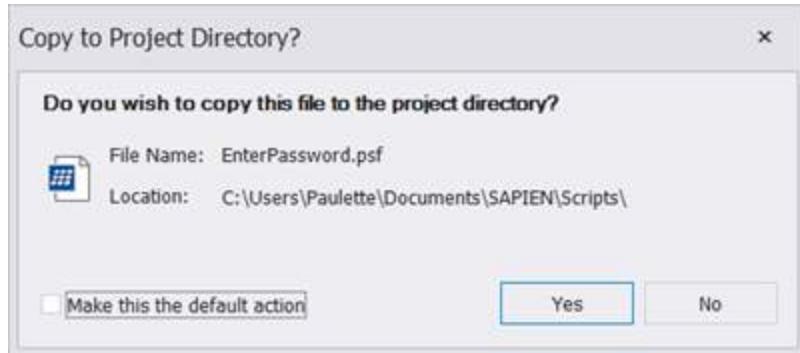
## Adding Files and Folders

### How to add an existing file

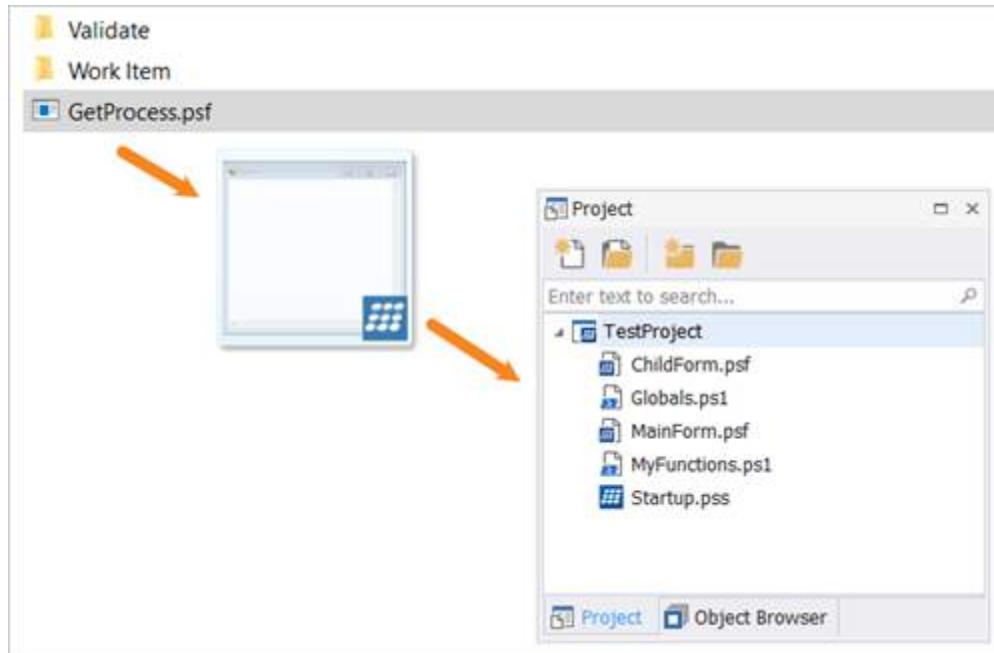
Click the Add Existing File button, select a file to add to the project, then click Open:



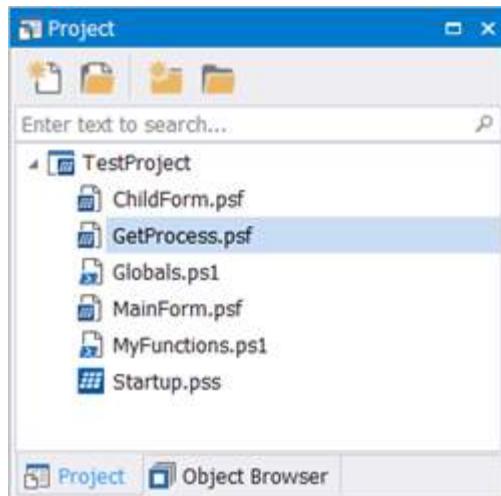
Select Yes if you want to create a copy of the file in the project directory:



You can also drag and drop files and folders into the Project from a file directory. Simply click and drag the desired file or folder, and then drop it in the project:



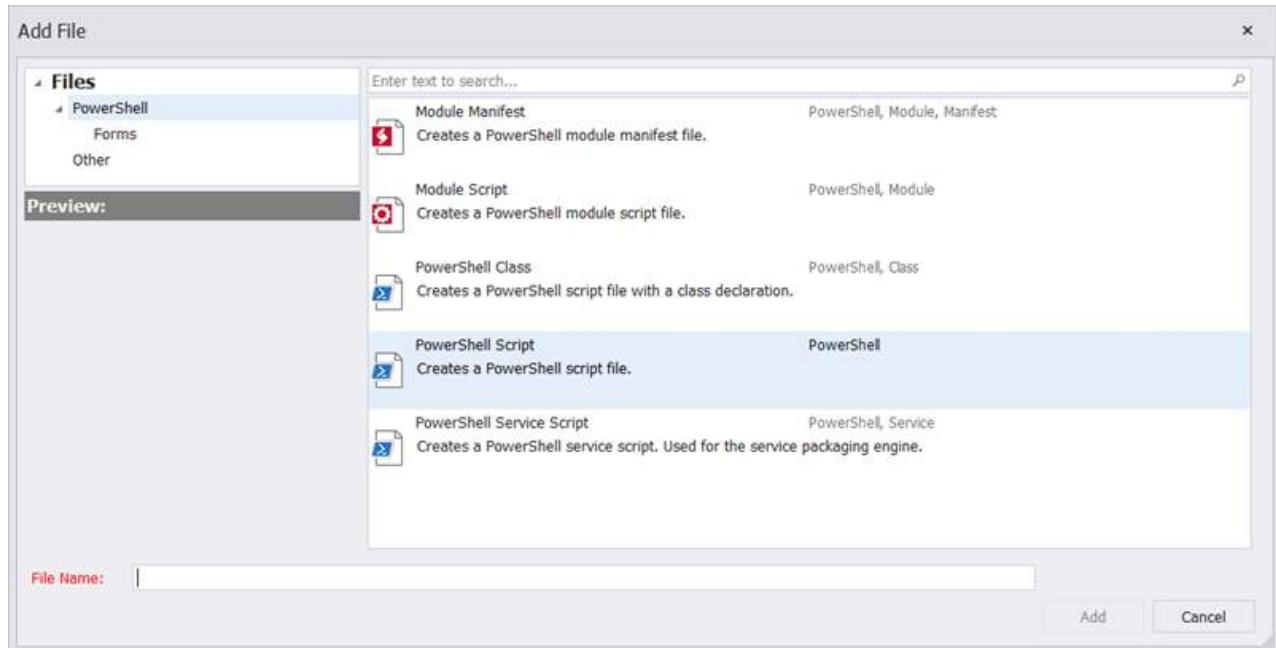
Select Yes if you want to create a copy of the file in the Project directory:



- i** Dragging and dropping a *folder* into the Project panel will automatically create a copy of the folder in the project directory.

#### How to add a new file

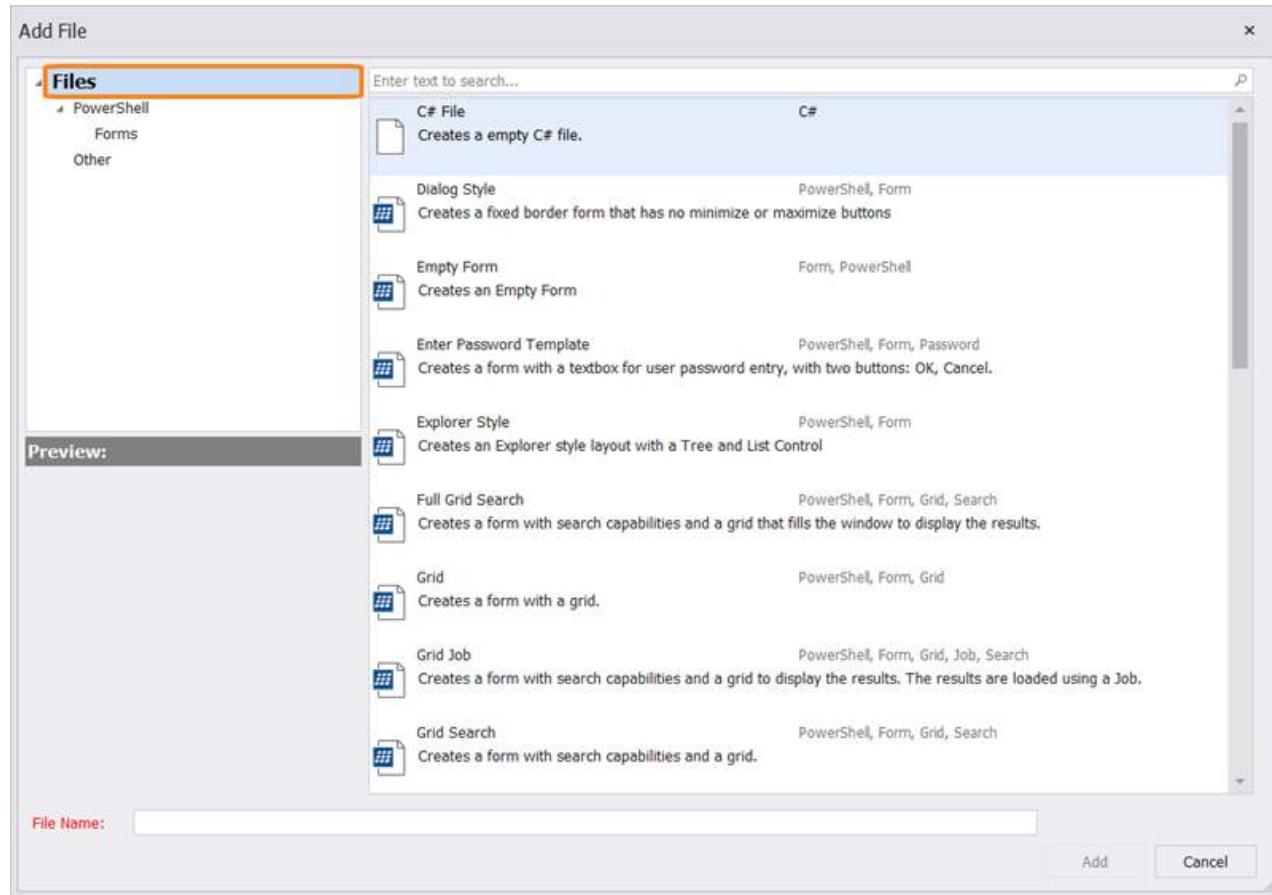
Click the New File button to launch the Add File dialog where you can select a file template:



The file templates are grouped into categories in the upper-left of the dialog. The PowerShell category lists all PowerShell script files, and the Forms sub-category contains the form templates:

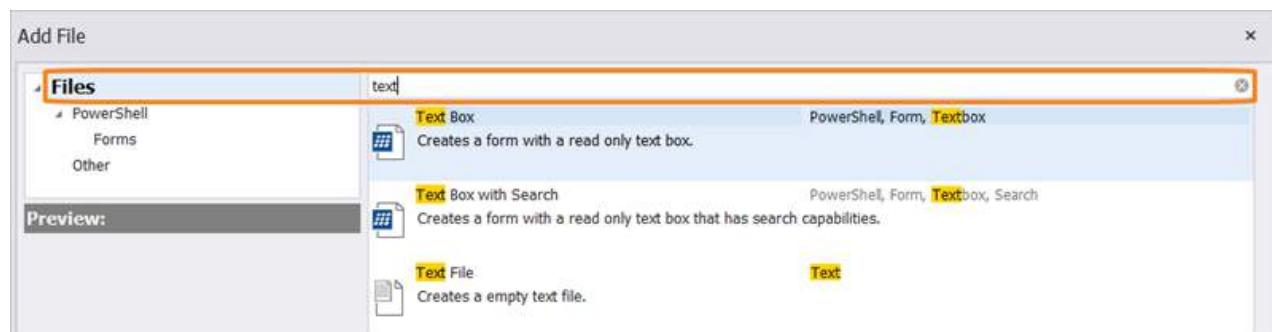


The **Files** category at the top lists all of the templates:



Use the search field to search the template titles and their tags:

- Select **Files** to search all available templates regardless of category.



- The small text located next to the template names are tags/keywords that describe the contents of the particular template:

**Full Grid Search** → **PowerShell, Form, Grid, Search**

Creates a form with search capabilities and a grid that fills the window to display the results.

The tags associated with the Full Grid Search template shown above indicate that this template is a PowerShell script that has a GUI with a Grid and has Search capabilities.

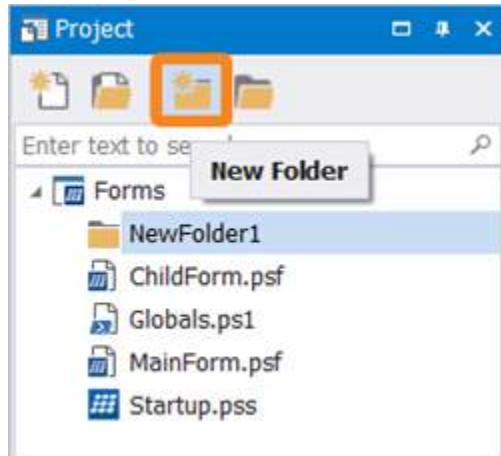
Tags can also be used to filter out file templates by language.

Once you have selected a file template, enter a File Name, and then click **Add** to add the new file to the project:

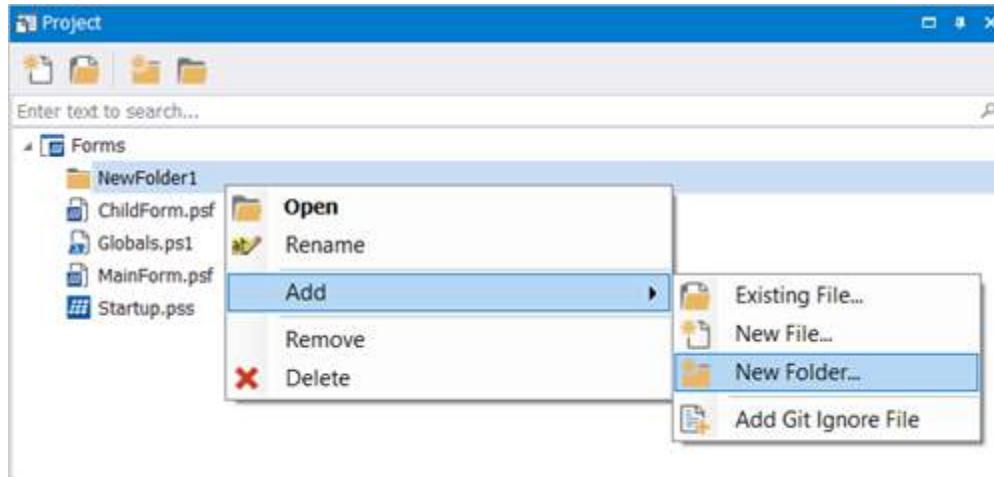


## How to add a folder

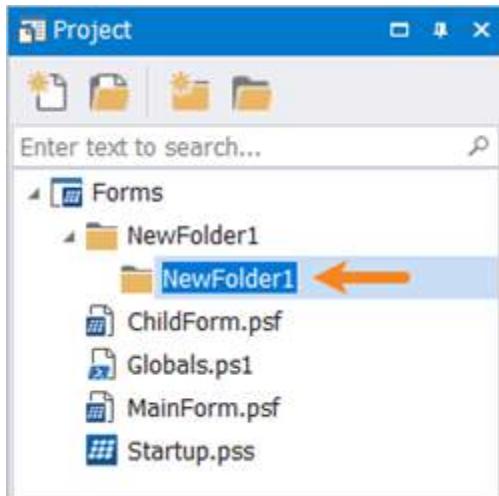
Click the **New Folder** button to add a new folder:



Or, you can add a new folder by right-clicking and selecting **Add > New Folder**:



The new folder will be created under the location highlighted when the folder was added:

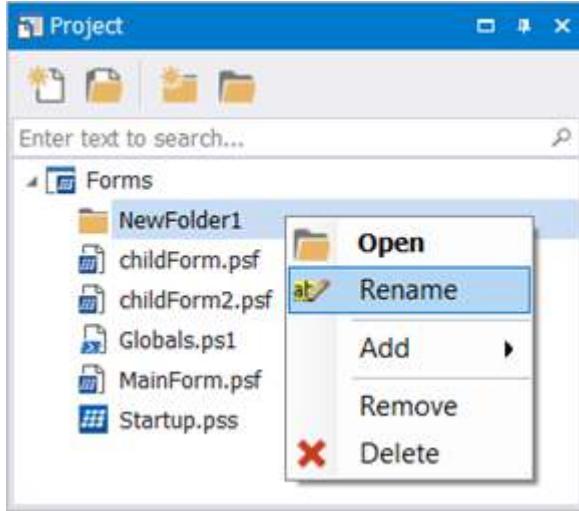


- i** Dragging and dropping a folder into the Project panel will *automatically create a copy of the folder in the project directory*.

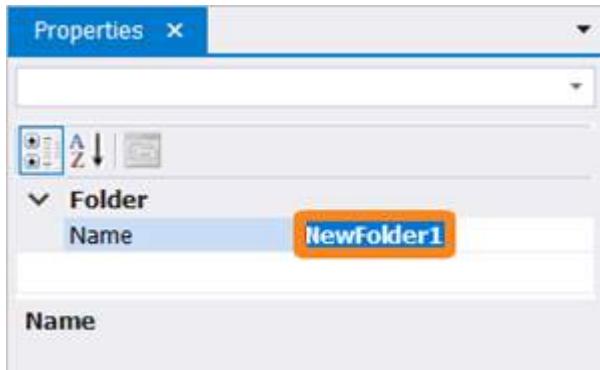
## Renaming Files and Folders

**!** To rename files you must use the Project panel instead of File Explorer, otherwise the project's references will point to incorrect paths.

You can rename files and folders by clicking twice on the file or folder name in the project panel, or by right-clicking and selecting **Rename**:



Alternately, you can rename a folder in the Properties panel by highlighting the name and typing a new name:

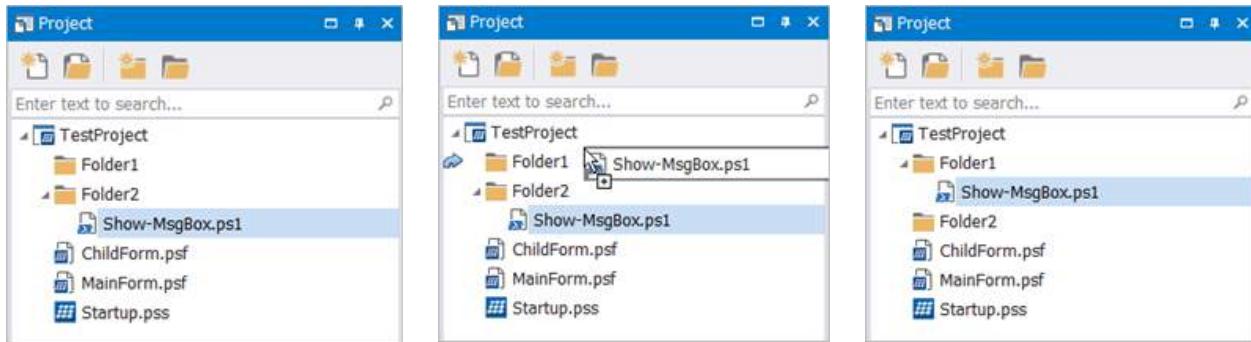


For more information on *Properties*, see [Properties Panel](#) [240]

## Moving Files and Folders

! To move files you must use the Project panel instead of File Explorer, otherwise the project's references will point to incorrect paths.

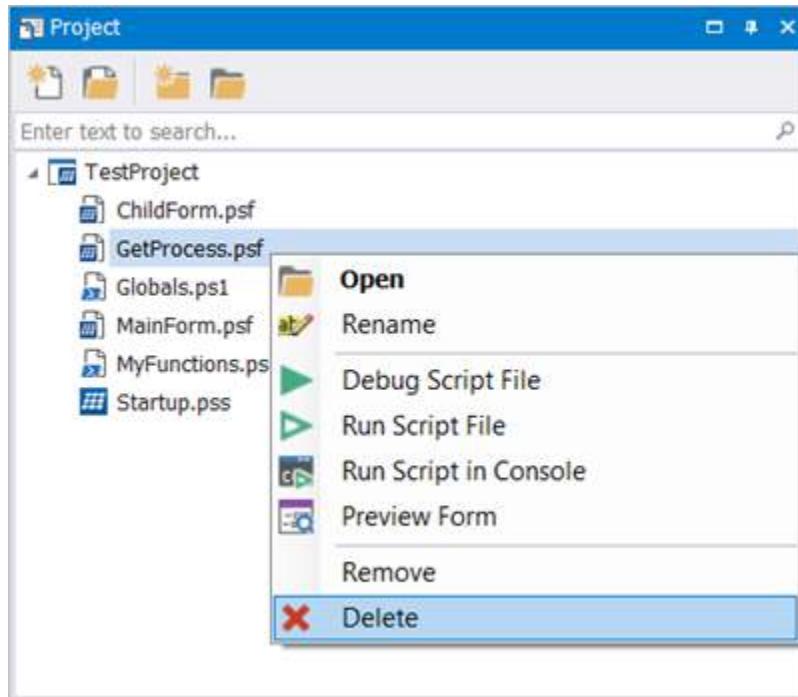
You can move files and folders within the Project panel by clicking the icon next to the name and dragging > dropping:



## Deleting Files and Folders

### How to delete a file

Right-click and select **Delete** or press the < **Delete** > key:



Select Yes to delete the file:

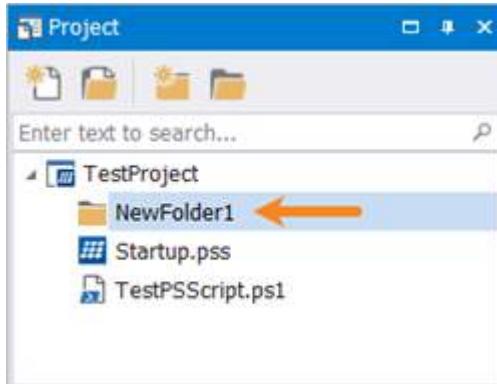


**i** The file is deleted, regardless of location:

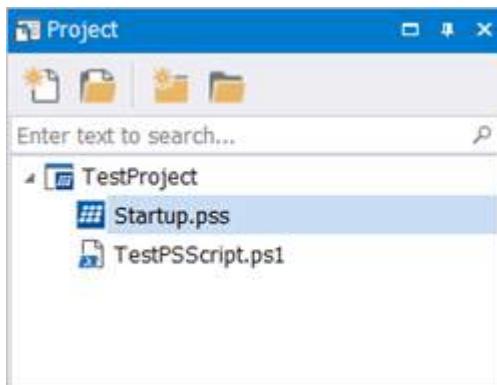
- If the file was *copied* into the Project, then the file will be deleted from the project directory.
- If the file was *not copied* into the Project, then the file will be deleted from the source location.

## How to remove a folder from the Project panel

Highlight the folder and press the < Delete > key (or right-click and select Remove):



The folder is removed from the Project panel:

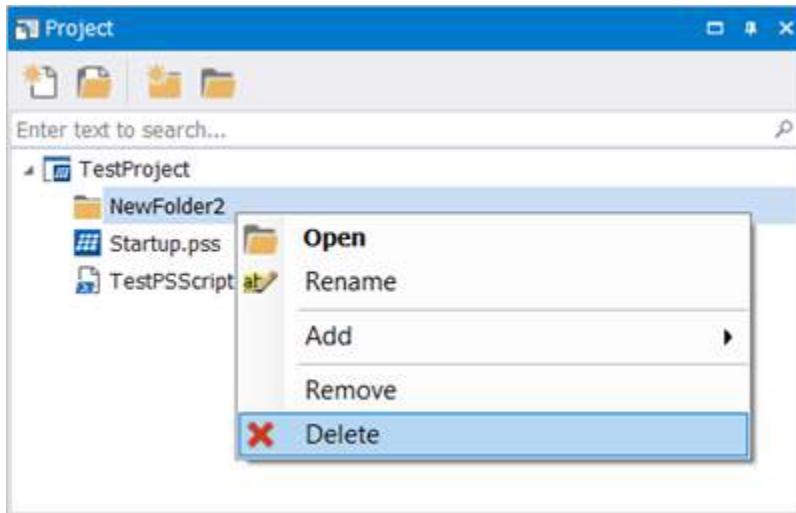


The folder remains in the project directory:

> Documents > SAPIEN > PowerShell Studio > Projects > TestProject	
Name	Type
TestProject.psproj	PSPROJ File
TestProject.psprojs	PSPROJS File
TestPSScript.ps1	Windows PowerShell Script
Startup.pss	PowerShell Studio Package Script Document
NewFolder1	File folder

#### How to remove a folder from the Project panel and delete it from the project directory

Right-click the folder and select **Delete**, then click **Yes** to confirm:



The folder will be deleted from the project directory *and* from the Project panel.

## 8.11 Properties Panel

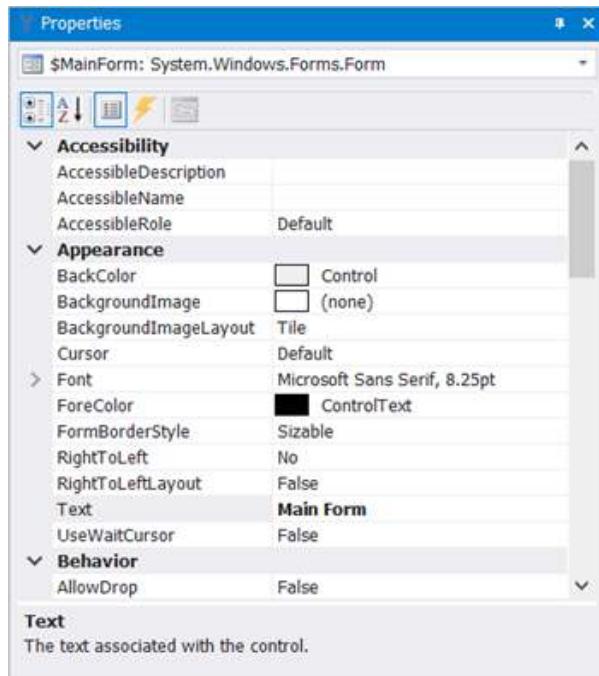
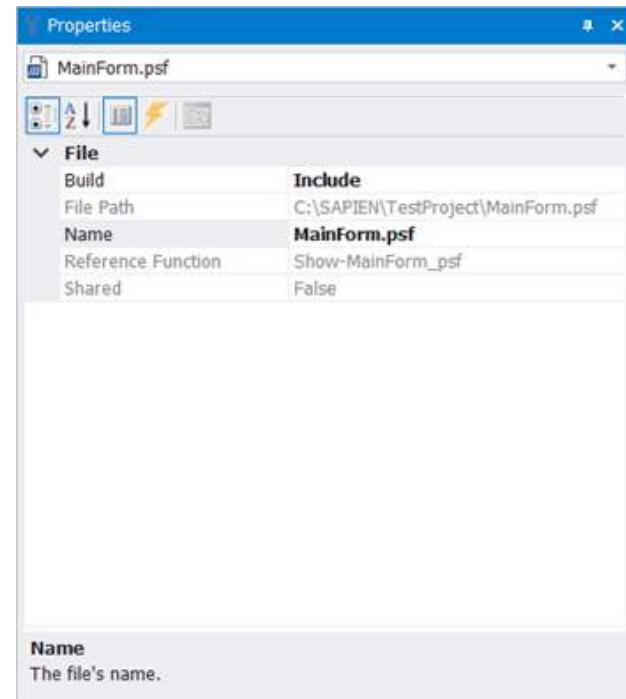
The Properties panel allows you to view and edit the control properties when working in the GUI Designer, and edit project settings and project file settings when working in a project. This topic provides an overview of the Properties panel and shows you how to [work with events](#) <sup>243</sup>.

### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***P***.

## Properties Panel Overview

The Properties panel allows you to view and edit the properties of any object selected in the [Forms Designer](#) <sup>152</sup> or the [Project panel](#) <sup>225</sup>.

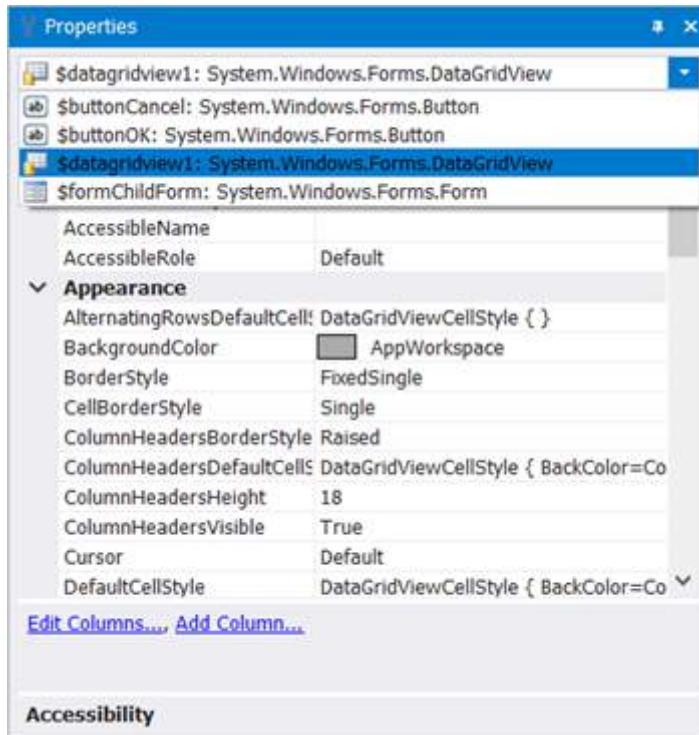
*GUI Designer Properties**Project File Properties*

- i** When you edit form controls in the Properties panel the changes are immediately reflected in the Designer.

There are a number of controls at the top of the Properties panel:

- **Object Picker Dropdown**

Allows you to switch between controls on a form:



- Properties - Sort Order Buttons

The pair of buttons on the left allow you to switch between viewing properties grouped into functional categories or listed alphabetically:

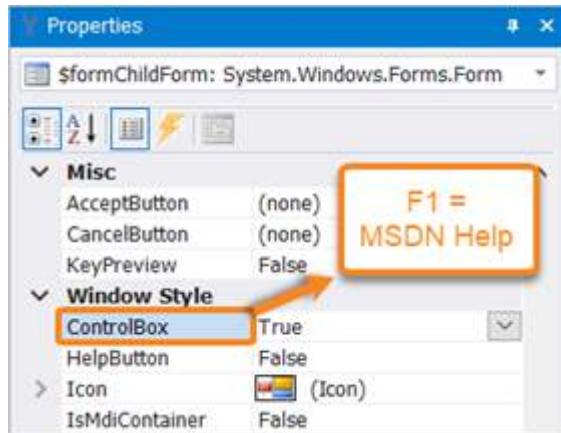


- Properties and Events View Buttons

The pair of buttons in the middle allow you to switch between looking at the properties an object supports or the events it can trigger:



- Pressing F1 when a control property is selected will open the related MSDN Help topic in a browser window:



## Working with Events

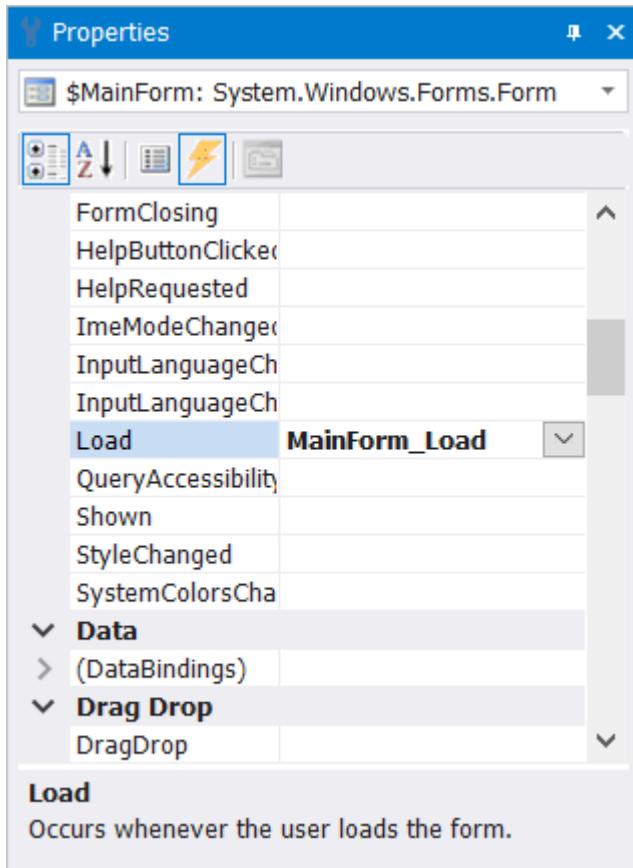
Event handlers can be created using the Properties panel.

### How to create an event handler

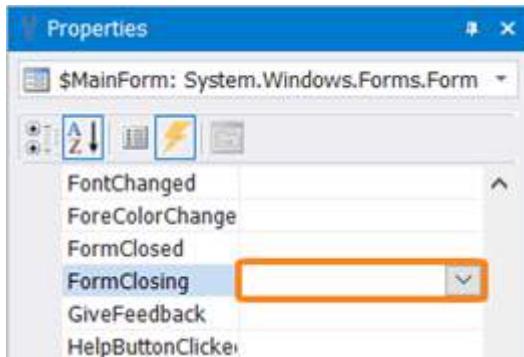
First select a control in the designer and access the Properties panel. Click on the lightning-bolt button to display the events that belong to the control.

- If multiple controls or project files are selected, only common properties and events are displayed. If you assign a value to a property, it is applied to all selected objects. If you assign an event to a control, the event will be assigned to all of the controls.

The following screenshot shows the events for a form. The Load event has been connected to a handler called Mainform\_Load:



To handle another event, simply double click in the blank cell next to the event:



PowerShell Studio will create an event handler named \$<object name>\_<event name> and insert it in the script at the caret position. For example, double-clicking in the blank cell next to FormClosing results in the following code being generated:

```
13 }  
14  
15 $MainForm_FormClosing=[System.Windows.Forms.FormClosingEventHandler]{  
16     #Event Argument: $_ = [System.Windows.Forms.FormClosingEventArgs]  
17     #TODO: Place custom script here  
18  
19 }  
20
```

👉 If you want to specify the name of the event handler you can type its name rather than double-clicking in the blank cell. PowerShell Studio will use the name you type to generate the event handler.

## 8.12 Snippets Panel

The Snippets panel allows you to view and manage preset and user-defined snippets (reusable text and code). This topic provides information about the Snippets panel, and shows you how to [manage snippets](#)<sup>247</sup> and work with [snippet folders](#)<sup>248</sup>.

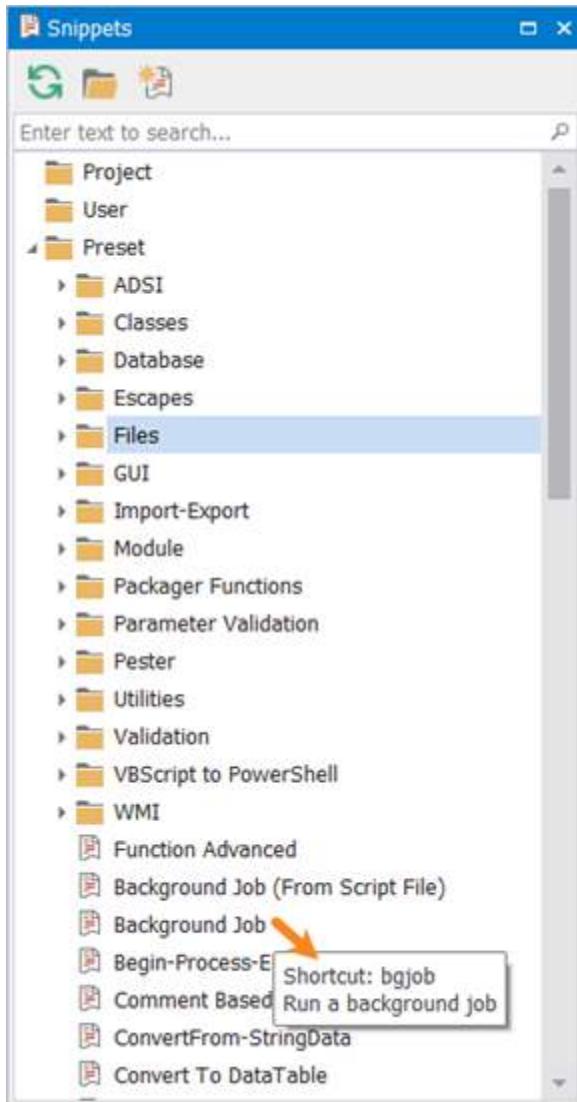
### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***S***.

### Snippets Panel Overview

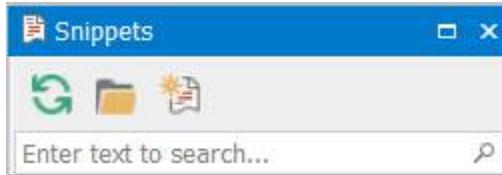
PowerShell Studio provides an extensive collection of snippets to help you complete common coding tasks quickly. Each snippet contains a block of code that can be easily added to a script.

Hover over a name in the Snippets panel to see the snippet Description and Shortcut:



## Snippets Panel - Buttons and Search

There are three buttons and a search box at the top of the Snippets panel:



From left to right:

- **Refresh**

Causes PowerShell Studio to rescan the snippets folder and refresh the Snippets panel.

- **Open**

Opens the currently selected snippets folder in File Explorer.

-  **New**

Launches the Snippet Editor to create your own snippet. It is also possible to edit and delete an existing snippet by right-clicking and choosing edit or delete respectively.

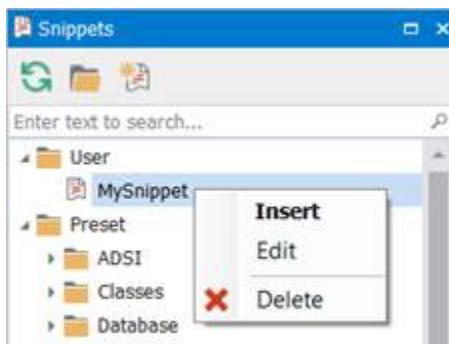
-  **Search**

Search for a snippet by typing the first few letters of a snippet in the search box. As you type, PrimalSense™ will give you a list of possible completions.

## Managing Snippets

---

Right-click on a snippet to display the following options:



- **Insert**

Use any of these options to add a snippet to a script. The snippet will be inserted at the current caret position in the code Editor:

- Right-click on a snippet and select **Insert**

**-OR-**

- Double-click a snippet

**-OR-**

Click the snippet and then press < **Enter** >

**-OR-**

- Drag a snippet and drop it in the code Editor.

- **Edit**

Opens the snippet in the [Snippet Editor](#) .

- **Delete**

Deletes the snippet.

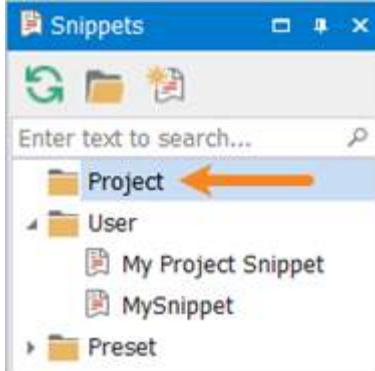
 You can drag-and-drop snippets within the Snippets panel.

## Snippet Folders

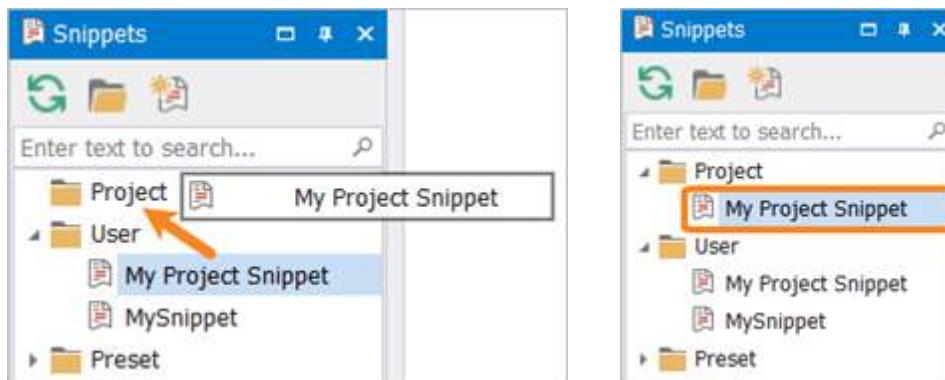
There are four snippet folders available in the following priority order: [Project](#), [User](#), [Custom](#), [Preset](#).

- **Project**

A 'Project' snippet folder will appear in the Snippets panel whenever you open a project:



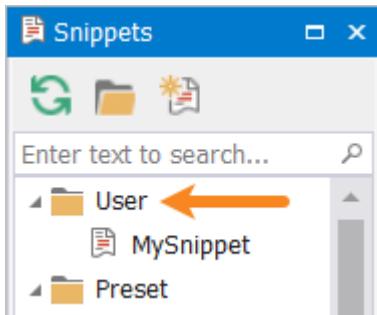
You can copy snippets to and from the project by dragging the snippet within the Snippets panel:



- i** A snippet can also be added to a project by creating or copying the snippet to the project's file directory, and PowerShell Studio will automatically display it in the Snippets panel when you open the project. It is not necessary to add the snippet to the project using the [Project panel](#).

- **User**

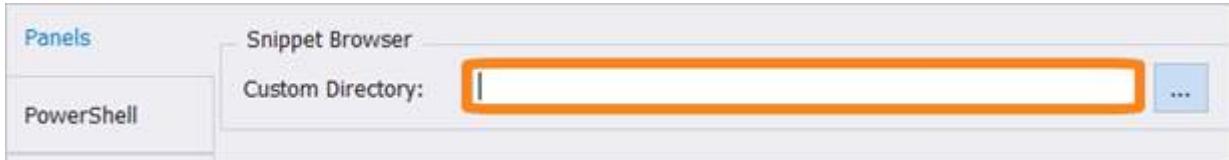
New snippets that you create will automatically be stored in the 'User' folder unless you change the folder path: %Users%\<user>\AppData\Roaming\SAPIEN\User Snippets\PowerShell:



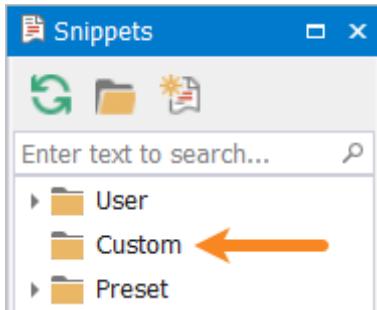
👉 To modify an existing snippet, copy it into the 'User' folder.

- **Custom**

You can add a custom directory to the Snippets panel, such as a shared network snippet repository, via **Home > Options > Panels**. Specify the folder path in the Custom Directory field:



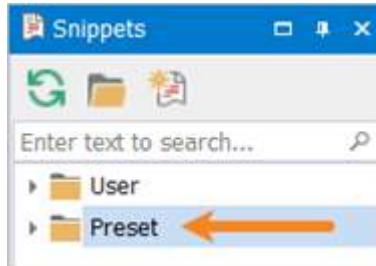
This folder is named 'Custom' in the Snippets panel, regardless of the folder path selected:



💡 After designating the custom directory folder path, you must refresh the Snippets panel to view the 'Custom' folder.

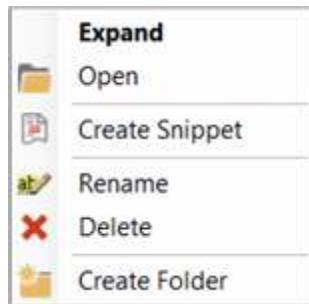
- **Preset**

The 'Preset' folder contains the snippets included with PowerShell Studio, and they are loaded from the following folder: %ProgramData%\SAPIEN\PowerShell Studio <year>\Snippets.



## Snippet Folders - Context Menu Options

Right-click on a snippet folder to access the following options:



- **Expand / Collapse**  
Expand or Collapse the subfolders.
- **Open**  
Open the folder in File Explorer.
- **Create Snippet**  
Create a snippet.
- **Rename**  
Rename the folder.
- **Delete**  
Delete the snippet folder and all of its contents.
- **Create Folder**  
Add a subfolder.

## 8.13 Toolbox Panel

The Toolbox panel displays Windows Forms controls and control sets that can be used when designing a PowerShell form in the [GUI Designer](#) 152.

- Controls are built in .NET controls.
- Control sets are custom controls built out of standard controls and custom scripts.

## Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***T***.

## Toolbox Panel - Buttons and Search

There are three buttons and a search box at the top of the Toolbox panel:

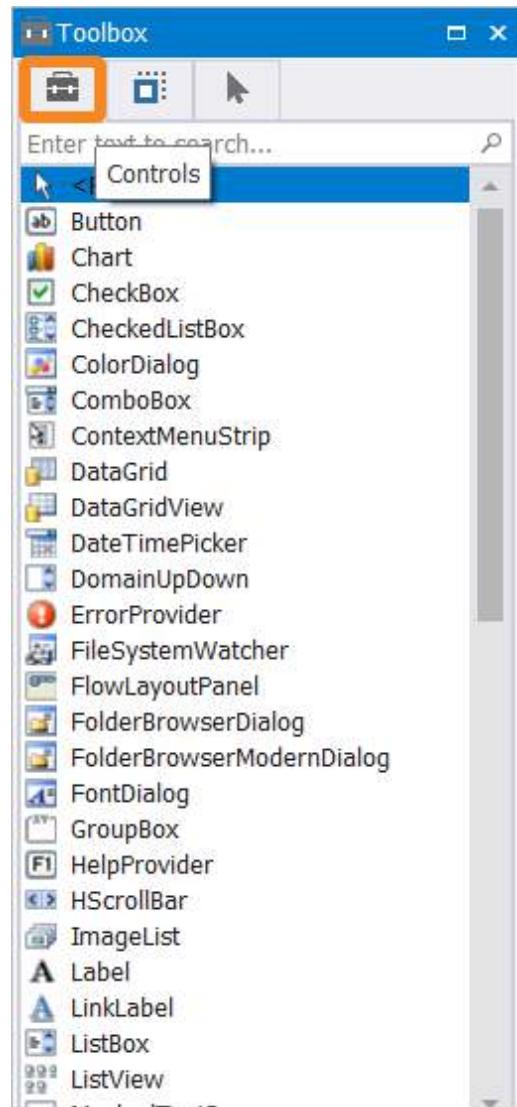
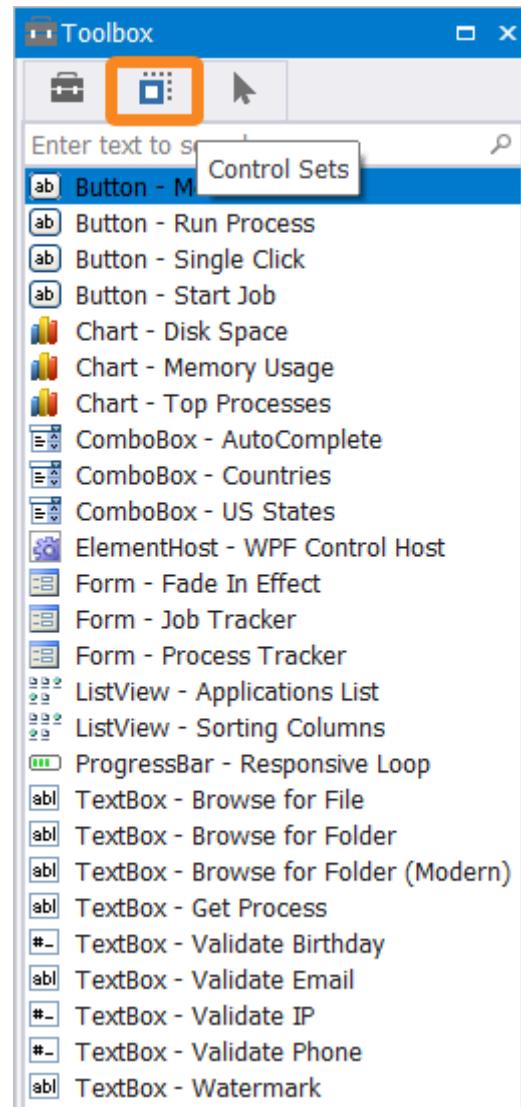


From left to right:

- **Controls**  
Displays a list of controls that can be added to a form.
- **Control Sets**  
Displays a list of control sets that can be added to a form.
- **Select**  
Displays the controls and control sets added to the current form that is open in the Forms Designer.
- **Search**  
Search for a control or control set by typing the first few letters in the search box. As you type, PrimalSense™ will give you a list of possible completions.

## Controls and Control Sets

- You must have a form open in the GUI Designer to activate the Controls and Control Sets options.

**Controls****Control Sets****Control - Context Menu Options**

Right-click on a control to display the following options:



- **Insert**  
Adds the control to the current form.

- **View in Object Browser**  
Focuses the object browser on the .NET control type.

- **MSDN Help**

Launches MSDN Help for this control in a web browser.

- **View Spotlight Article**

Launches tutorial content for this control on the SAPIEN web site.

## Control Set - Context Menu Options

Right-click on a control set to display the following option:



- **Insert**

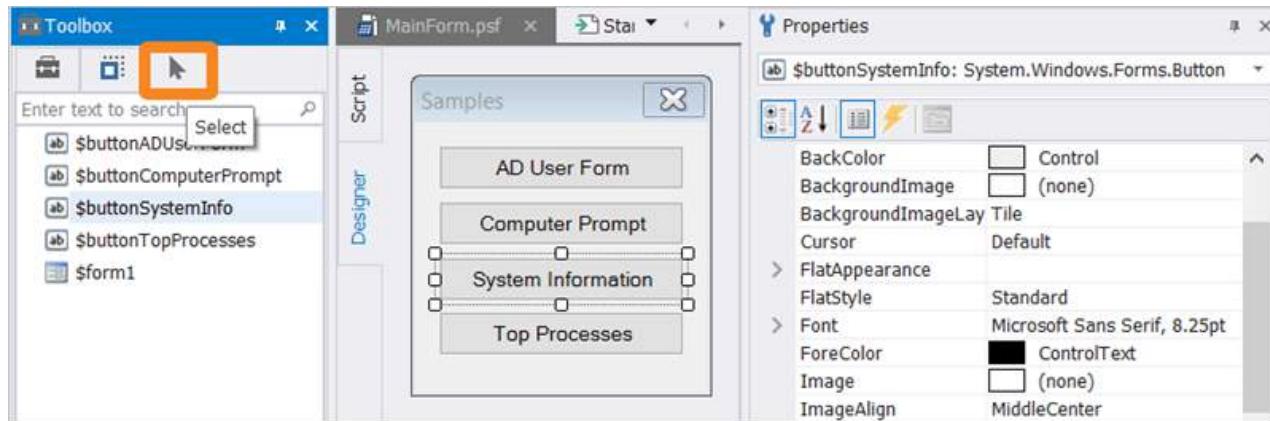
Adds the control to the current form.

There are three ways to add a control or control set to a form:

- **Drag and drop** the control or control set on the form.
- **Double-click** the control or control set.
- **Right-click** on the control or control set, then select **Insert**.

## Select Tool

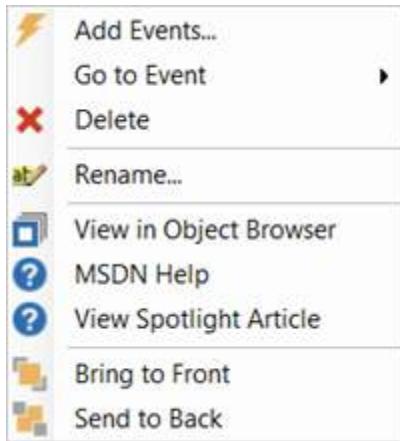
The Select tab is on the top of the Toolbox panel, to the right of the Control Sets tab. Use the Select tool to select a control, which selects the control in the Designer and displays the control properties:



Use **Shift+Click** or **Ctrl+Click** to select multiple controls. When multiple controls are selected, the Properties panel displays only the properties that the selected controls have in common.

## Select Tool - Context Menu Options

Right-click on a control in the Select list to display the following options:



- **Add Events...**

Opens the Add Events dialog to select events to add to the control.

- **Go to Event**

Displays the events that are wired to the control. When clicked, it will go to the event in the Editor.

- **Delete**

Deletes the control from the form.

- **Rename**

Opens the Rename object dialog.

- **View in Object Browser**

Focuses the object browser on the .NET control type.

- **MSDN Help**

Launches MSDN Help for this control in a web browser.

- **View Spotlight Article**

Launches tutorial content for this control on the SAPIEN web site.

- **Bring to Front**

Sends the control to the top/front of the form.

- **Send to Back**

Sends the control to the bottom/back of the form.

👉 Use the **Bring to Front** context menu option to send the control to the top of the form without having to find the control in the Designer.

## 8.14 Tools Output Panel

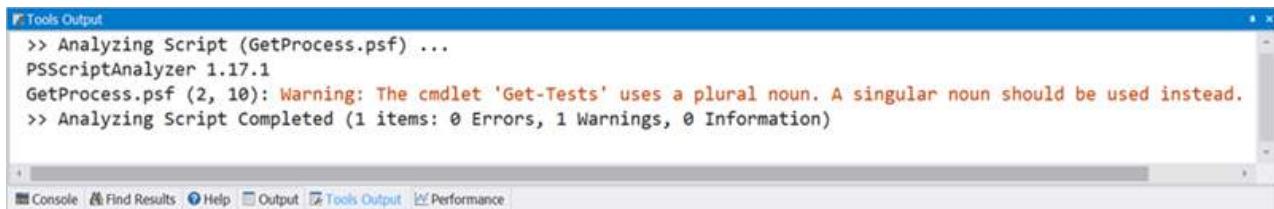
The Tools Output panel displays output from external tools. When debugging, the Tools Output panel displays breakpoint notifications and post mortem messages.

### Keyboard Shortcut

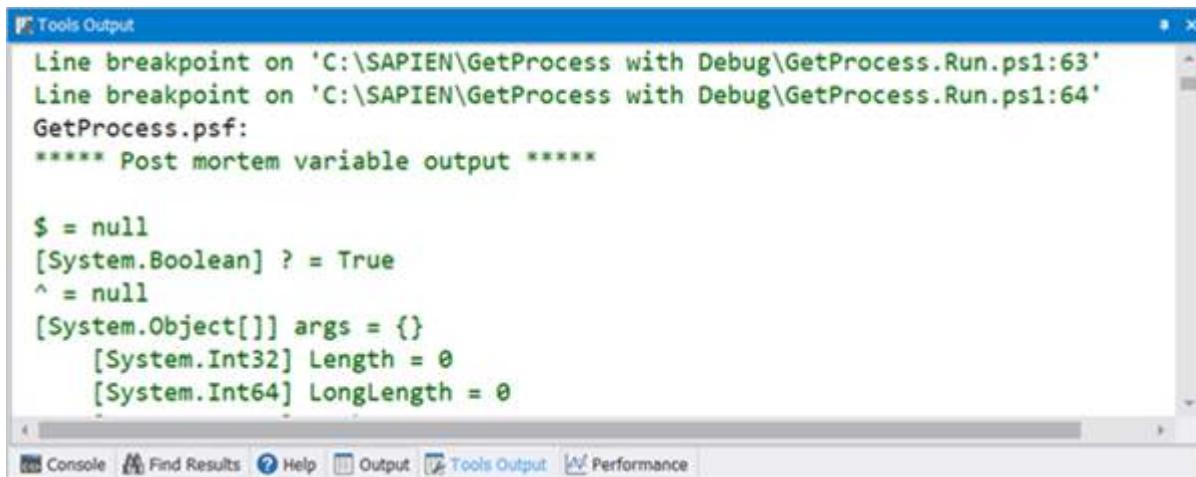
Press ***Ctrl + Alt + P***, release, then press ***L***.

## Tools Output Panel Overview

The Tools Output panel displays output from any tool, such as the *Universal Version Control* system, *Custom Menu*, or the *PSScriptAnalyzer Module*:

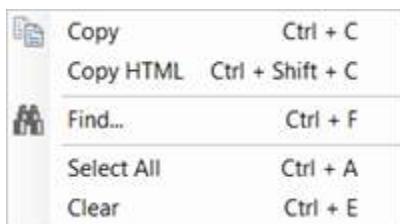


When debugging, breakpoint notification and the post mortem state of the variables are displayed:



## Tools Output Panel - Context Menu Options

Right-click in the Tools Output panel to display the following options:



- **Copy (Ctrl+C)**

Copy highlighted text to the clipboard.

- **Copy HTML (Ctrl+Shift+C)**

Copied highlight code, including color coding and formatting.

- **Find (Ctrl+F)**  
Search the text in the output panel.
- **Select All (Ctrl+A)**  
Select all of the text in the output panel.
- **Clear (Ctrl+E)**  
Clear the output panel.

## 8.15 Variables Panel

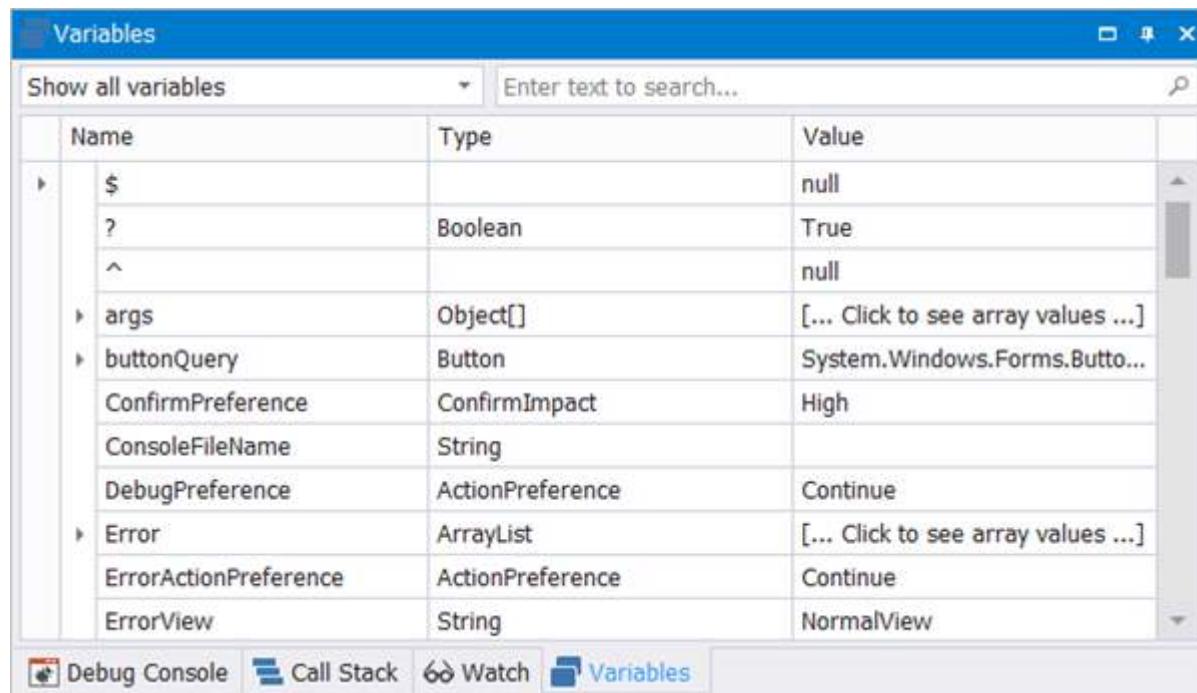
The Variables panel lists all variables and values in the current scope during a breakpoint when debugging.

### Keyboard Shortcut

Press **Ctrl + Alt + P**, release, then press **V**.

### Variables Panel Overview

The Variables panel shows the current values of all PowerShell variables during a debugging session at a breakpoint:

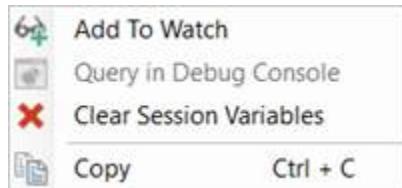


The screenshot shows the Windows PowerShell ISE Variables panel. The title bar says "Variables". The main area is a grid table with columns "Name", "Type", and "Value". The "Name" column contains variable names like \$, ?, ^, args, buttonQuery, ConfirmPreference, ConsoleFileName, DebugPreference, Error, ErrorActionPreference, and ErrorView. The "Type" column shows their types such as Boolean, Object[], Button, ConfirmImpact, String, ActionPreference, ArrayList, and ActionPreference. The "Value" column displays the current value for each variable. A search bar at the top right says "Enter text to search...". At the bottom, there are tabs for "Debug Console", "Call Stack", "Watch", and "Variables", with "Variables" being the active tab.

Name		Type	Value
\$			null
?	Boolean		True
^			null
args	Object[]		[... Click to see array values ...]
buttonQuery	Button		System.Windows.Forms.Button...
ConfirmPreference	ConfirmImpact		High
ConsoleFileName	String		
DebugPreference	ActionPreference		Continue
Error	ArrayList		[... Click to see array values ...]
ErrorActionPreference	ActionPreference		Continue
ErrorView	String		NormalView

### Variables Panel - Context Menu Options

Right-click in the Variables panel to display the following options:



- **Add To Watch**

Add the variable to the [Watch panel](#).

- **Query in Debug Console**

Display the variable details in the Debug Console panel during a debugging breakpoint.

- **Clear Session Variables**

Clear the Variables panel.

- **Copy (Ctrl+C)**

Copy the selected variable to the clipboard.

## Variables - Filter and Search

### How to filter and search for variables

Use the drop-down menu to show All variables, User variables, or PowerShell variables:



- **Show all variables**

Displays all the variables in the current debug session.

- **User variables only**

Displays user defined variables only.

- **PowerShell variables only**

Displays the PowerShell built-in variables.

### How to filter variables by name

Type the variable into the Search box:

The screenshot shows two instances of the 'Variables' panel from a debugger. The top instance has a search bar containing 'Enter text to search...' and displays a table with three rows: one for a variable '\$' with type 'null' and value 'null'; and two for variables '?' with type 'Boolean' and value 'True'. The bottom instance has a search bar containing 'textbox' and displays a table with two expanded entries: 'textbox1' (Type: TextBox, Value: System.Windows.Forms.TextBox...) and 'richtextbox1' (Type: RichTextBox, Value: System.Windows.Forms.RichTextBox...).

Name	Type	Value
\$	null	null
?	Boolean	True

Name	Type	Value
textbox1	TextBox	System.Windows.Forms.TextBox...
richtextbox1	RichTextBox	System.Windows.Forms.RichTextBox...

## Variables - Object Properties

In addition to displaying the values of all variables, the Variables panel also lets you examine the properties of objects.

### How to expand the object properties

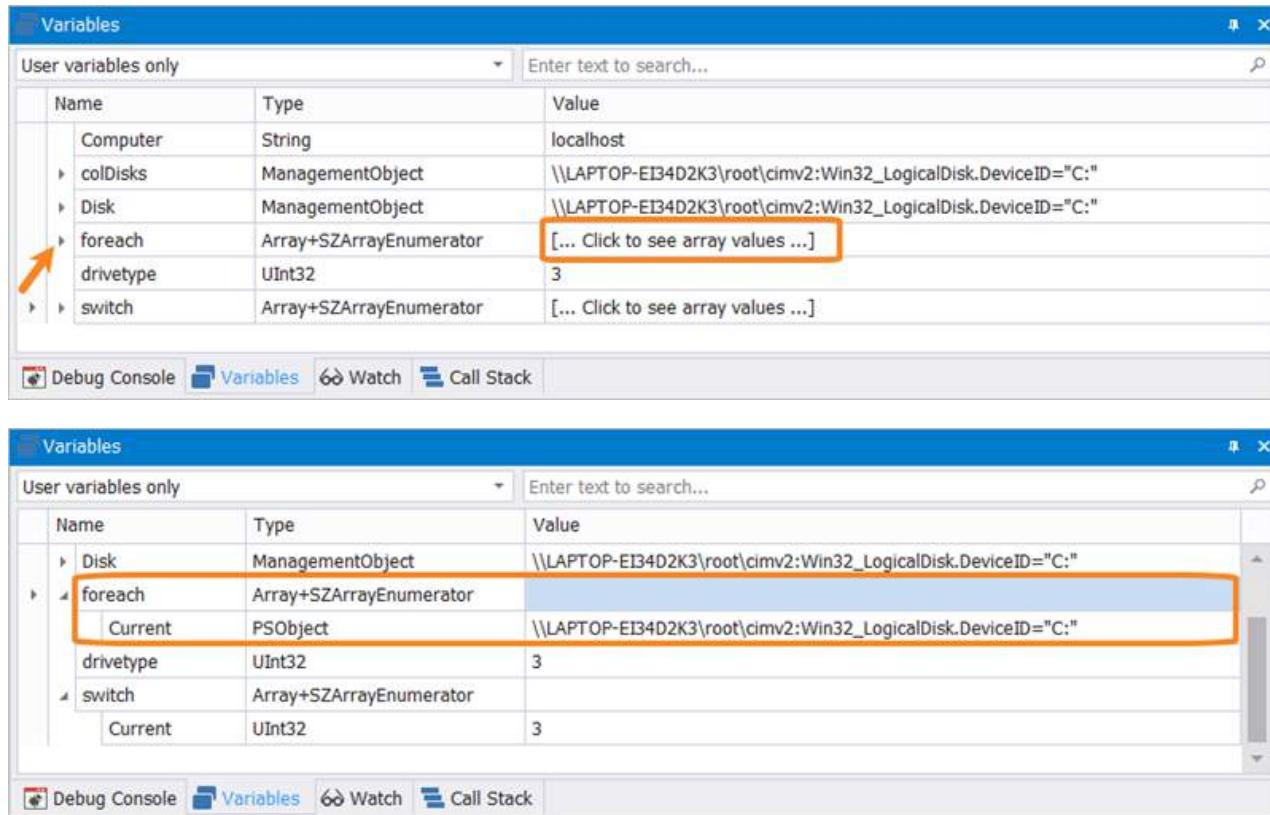
Double-click a variable name or click on the arrow (>) next to a variable name:

The screenshot shows the 'Variables' panel with the variable 'buttonQuery' selected. An arrow icon to the left of the variable name indicates it can be expanded. The expanded view shows five properties: AutoSizeMode (AutoSizeMode), DialogResult (DialogResult), AutoEllipsis (Boolean), AutoSize (Boolean), and BackColor (Color). The 'buttonQuery' row is highlighted in blue.

Name	Type	Value
buttonQuery	Button	System.Windows.Forms.Bu...
AutoSizeMode	AutoSizeMode	GrowOnly
DialogResult	DialogResult	None
AutoEllipsis	Boolean	False
AutoSize	Boolean	False
BackColor	Color	Color [Control]

### How to expand the array values for a variable during a debugging breakpoint

Click the arrow (>) on the left-side of the variable row, or double-click anywhere on the variable row:



The screenshot shows two instances of the Variables panel. The top instance displays a list of variables with their types and values. The 'foreach' variable is expanded, revealing its current item, 'Current'. The bottom instance shows a similar list, also with the 'foreach' variable expanded to show its current item, 'Current'. Both the 'foreach' row and the 'Current' row in both panels are highlighted with a red box.

Name	Type	Value
Computer	String	localhost
colDisks	ManagementObject	\LAPTOP-EI34D2K3\root\cimv2:Win32_LogicalDisk.DeviceID="C:"
Disk	ManagementObject	\LAPTOP-EI34D2K3\root\cimv2:Win32_LogicalDisk.DeviceID="C:"
foreach	Array+SZArrayEnumerator	[... Click to see array values ...]
drivetype	UInt32	3
switch	Array+SZArrayEnumerator	[... Click to see array values ...]

Name	Type	Value			
Disk	ManagementObject	\LAPTOP-EI34D2K3\root\cimv2:Win32_LogicalDisk.DeviceID="C:"			
foreach	Array+SZArrayEnumerator	<table border="1"> <tr> <td>Current</td> <td>PSObject</td> <td>\LAPTOP-EI34D2K3\root\cimv2:Win32_LogicalDisk.DeviceID="C:"</td> </tr> </table>	Current	PSObject	\LAPTOP-EI34D2K3\root\cimv2:Win32_LogicalDisk.DeviceID="C:"
Current	PSObject	\LAPTOP-EI34D2K3\root\cimv2:Win32_LogicalDisk.DeviceID="C:"			
drivetype	UInt32	3			
switch	Array+SZArrayEnumerator				
Current	UInt32	3			

## 8.16 Watch Panel

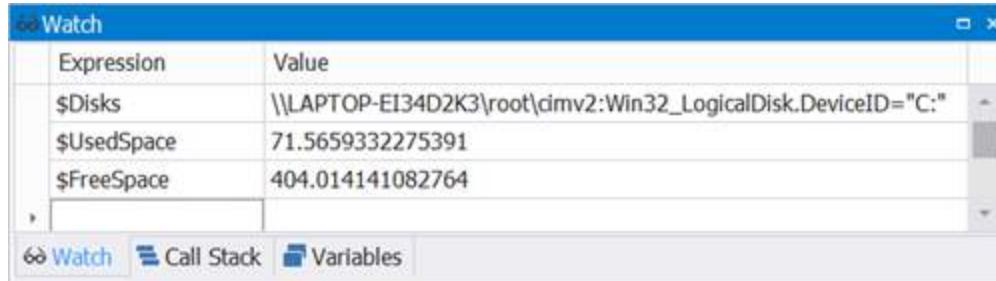
The Watch panel displays the values of variables and expressions that you define when debugging.

### Keyboard Shortcut

Press ***Ctrl + Alt + P***, release, then press ***W***.

### Watch Panel Overview

The Watch panel allows you to choose the variables you want to monitor during debugging:

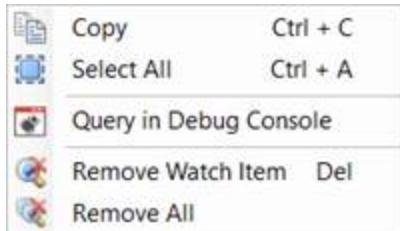


The screenshot shows the Watch panel with three entries. The first entry is '\$Disks' with a value of '\\LAPTOP-EI34D2K3\root\cimv2:Win32\_LogicalDisk.DeviceID="C:"'. The second entry is '\$UsedSpace' with a value of '71.5659332275391'. The third entry is '\$FreeSpace' with a value of '404.014141082764'. The tabs at the bottom of the panel are 'Watch' (selected), 'Call Stack', and 'Variables'.

Expression	Value
\$Disks	\LAPTOP-EI34D2K3\root\cimv2:Win32_LogicalDisk.DeviceID="C:"
\$UsedSpace	71.5659332275391
\$FreeSpace	404.014141082764

## Watch Panel - Context Menu Options

Right-click in the Watch panel to display the following options:



- **Copy (Ctrl+C)**

Copy the selected item to the clipboard.

- **Select All (Ctrl+A)**

Select all fields and values in the Watch panel.

- **Query in Debug Console**

Display the watched item details in the [Debug Console](#) (208) during a debugging breakpoint.

- **Remove Watch Item (Delete)**

Clear the item from the Watch panel.

- **Remove All**

Clear all items from the Watch panel.

## Using the Watch Panel

### How to add a new variable to the Watch panel

- Type its name in the next free slot in the Expression column.

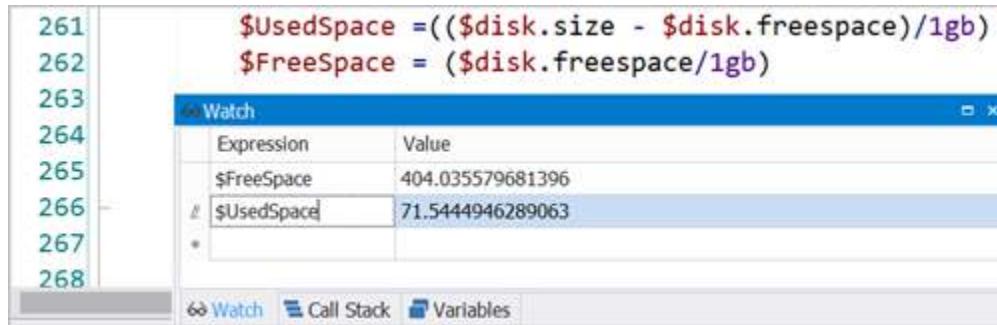
-OR-

- Highlight the variable in your code, then right-click and select Add to Watch.

-OR-

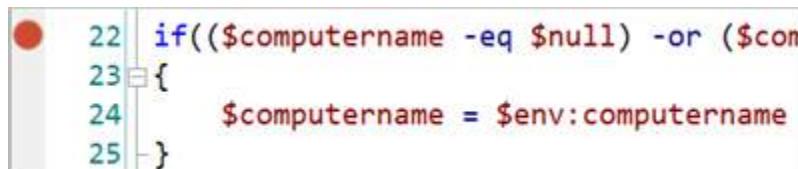
- Highlight the variable in your code, then click to drag and drop the variable in the Watch panel.

You can also monitor the values of object properties or calculated expressions:

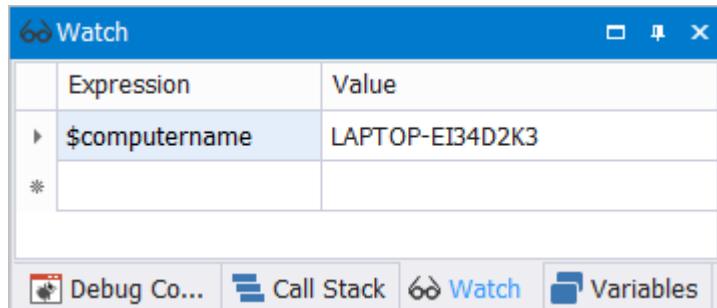


### Evaluating Expressions

The debugger can evaluate complex expressions to help you see what results your script is generating. To evaluate an expression while a script is paused, highlight the expression in the code window:



Drag the expression to the Watch panel. The expression is immediately evaluated and the result is displayed. The expression will be re-evaluated each time a line of script is executed allowing you to continually view the expression's result.



Any number of expressions can be added to the list. To remove an expression, select it in the list and press **Delete**.

## 9 Projects

PowerShell Studio *projects* facilitate the grouping of related files and settings. For example, a PowerShell utility might consist of several different Forms and Scripts. PowerShell Studio allows you to keep these files together as a project, and then use the built-in packager to create an executable file which includes everything in the package. Using a project instead of a single file script makes it easier to manage additional content, and allows you to organize your script into individual script files to make your code more manageable.

Other valuable uses for projects include management of the development workflow. With PowerShell Studio you can develop the project in a "sandbox" on your local machine, and then deploy the completed, tested, and debugged files as a single unit to a "live" production environment.

### 9.1 Project Templates

PowerShell Studio contains predefined templates for various project types.

#### 9.1.1 Available Project Templates

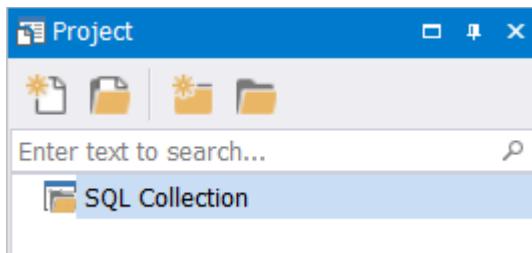
This topic explains the project template options.

### Project Templates

---

#### Collection Project

A *collection* project template allows you to group and deploy independent script files in an organized manner. The group of files typically consist of, but are not limited to, ps1 script files. A *collection* project is useful when you have various ps1 scripts that dot source each other.



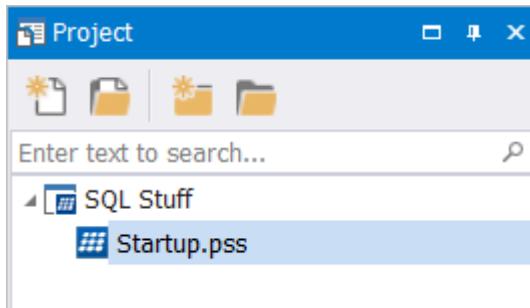
All files within the project are considered 'content'. There is no project entry point (`Startup.pss` - **project startup script**) because the project consists of individual files.

[Learn more about creating a Collection project](#) [268]

---

#### Empty Project

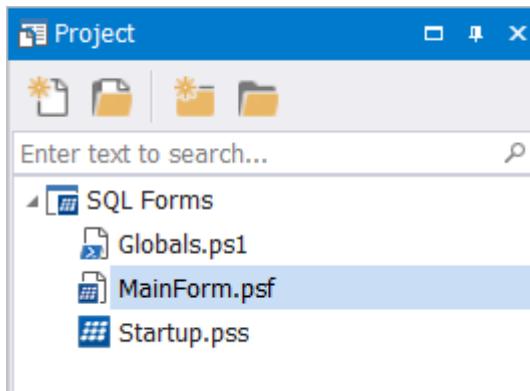
The *empty* project template creates an empty project for a script application.



The *empty* project template creates a basic project with an entry point. The empty project contains a single file called `Startup.pss` that runs when your project is executed. `Startup.pss` contains a `Main` function which serves as the entry point to the project / script, and is a good place do any preparatory work before calling other scripts in the project.

## Form Project

The *form* project template is used to create a GUI script with additional scripts and files.



A *form* project contains three files:

- **Globals.ps1**

A script file containing functions and variables that will be available throughout your project. Anything you define in this script will become global to the project.

- **MainForm.ps1**

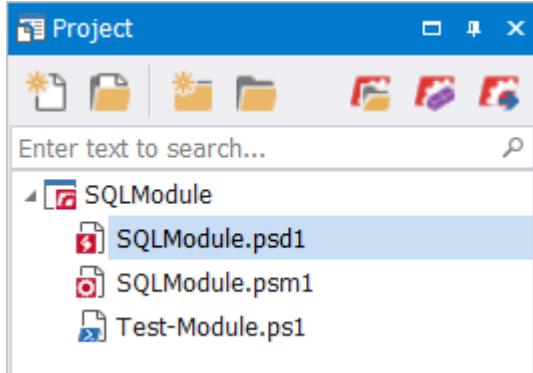
An empty form in which you build the GUI for your script.

- **Startup.pss**

The script that runs when your project is executed. `Startup.pss` contains a `Main` function which serves as the entry point to the project / script.

## Module Project

A *module* project is used for creating a PowerShell script module. The *module* project template is useful for creating packaged, reusable utilities that can be installed anywhere they are required.



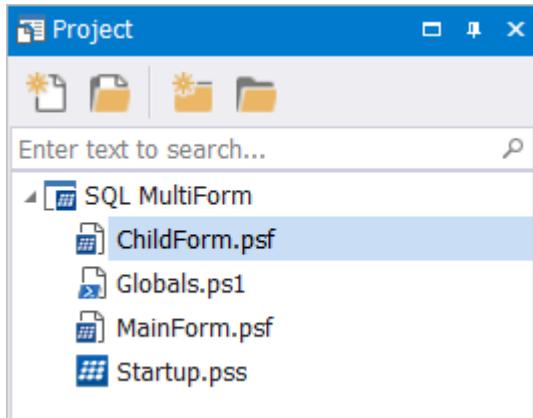
A *module* project contains three files:

- <*ProjectName*>.psd1 \*  
The manifest file for your module.
  - <*ProjectName*>.psm1 \*  
The PowerShell script code for your module.
  - **Test-Module.ps1**  
The PowerShell script code for testing your module.
- \* The file names are the same as the project name.

[Learn more about creating a Module project](#) [271]

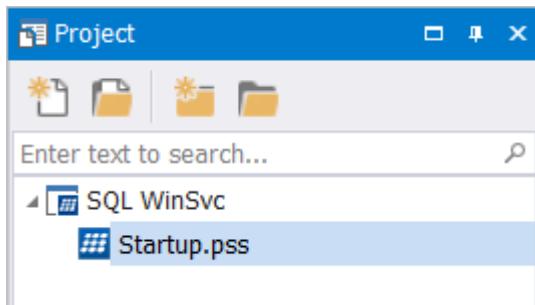
## Multi-Form Project

A *multi-form* project is the same as a form project with the addition of another form called ChildForm.psf.



## Windows Service Project

A Windows Service project template allows you to create a Windows PowerShell service project, and is used for the service packaging engine.



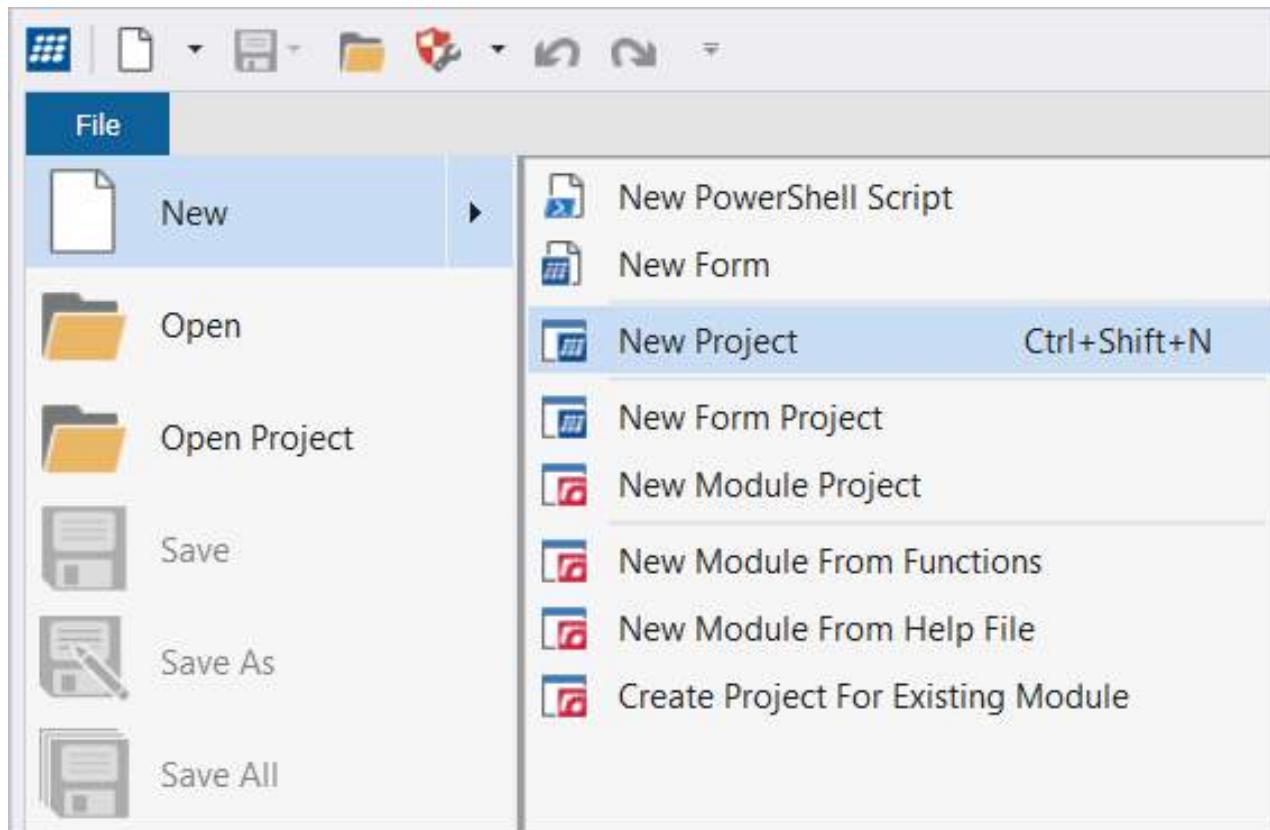
### 9.1.2 Creating a Project

This topic shows you how to create a new project using the New Project template.

## How to create a new project

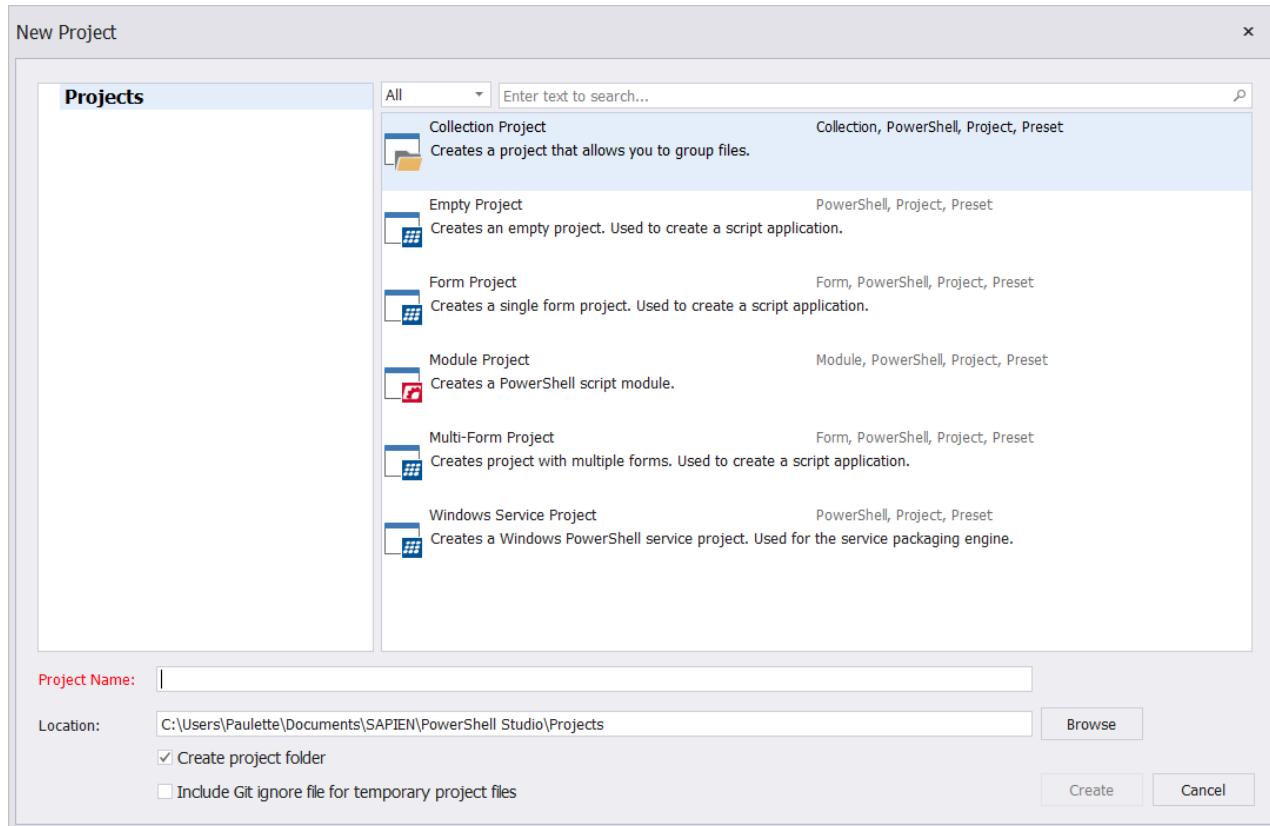
### To create a project

Select the **File** tab > **New** >**New Project** (*Ctrl+Shift+N*):

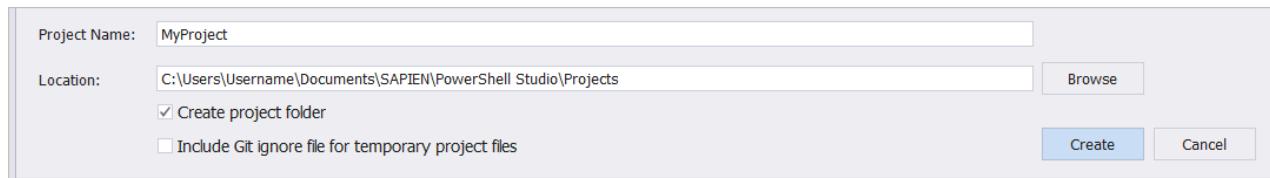


👉 You can go directly to some project templates from the **File > New** menu, such as **New Form Project** or **New Module Project**.

Select a template in the New Project dialog:



After selecting a template, complete the following information:



- **Project Name**

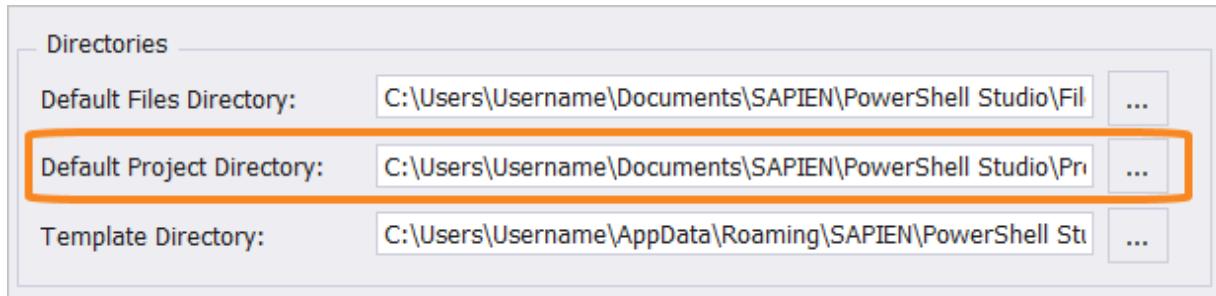
A name for your project.

- **Location**

The folder to store your project files in.

The default location for project files is: %Users%\<user>/Documents\SAPIEN\PowerShell Studio\Projects>

This path can be changed in **Home > Options > General > Directories > Default Project Directory**:



- **Create project folder**

Select this option to store your project files in a new sub-folder in the 'Projects' folder. Uncheck this option if you want to store your project file in an existing folder.

 Checking **Create Project Folder** will ensure that your project files are stored in their own sub directory in the 'Projects' folder, rather than mixing them together with files from other projects.

- **Include Git ignore file for temporary project files**

When checked, PowerShell Studio will create a .gitignore file for Git source control that filters any temporary project files.

Next, click **Create** to create the project. PowerShell Studio will create all of the files specified in the template and display them in the [Project panel](#) [225].

### 9.1.3 Collection Project

The *collection* project allows you to group and deploy independent script files in an organized manner.

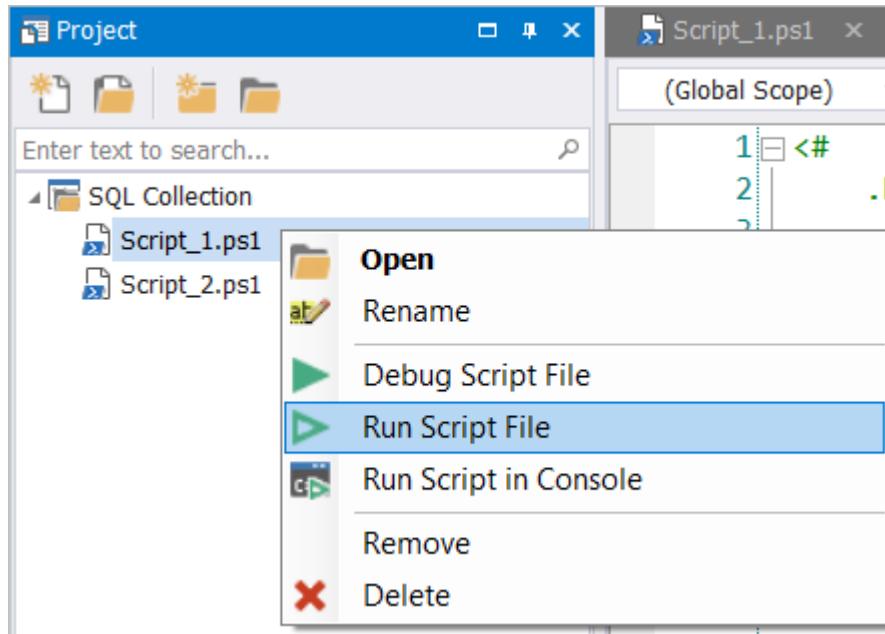
#### About Collection Projects

If you have various ps1 scripts that dot source each other, a Collection project will allow you to:

- Manage multiple files.
- Leverage PrimalSense support for dot sourced files.
- Apply rename refactoring to all of your files.

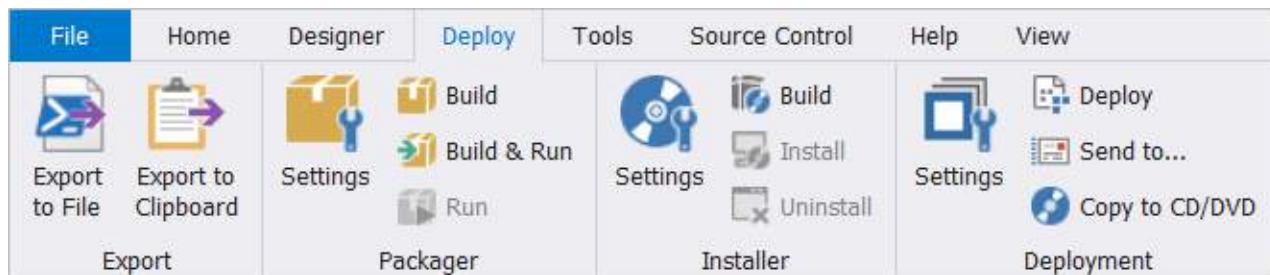
When a Collection project is created there is no Startup.pss file; there is no entry point in the project because the project contains individual files. All files within the project are considered 'content'.

You can run each file individually by right-clicking on a file in the Project panel, or from the ribbon (**Home** tab > **Run** menu > **Run options**):



## Collection Project Deployment

Deployment is handled on a project basis. You can deploy all the project files to a single destination or create an install that includes all of the project files.

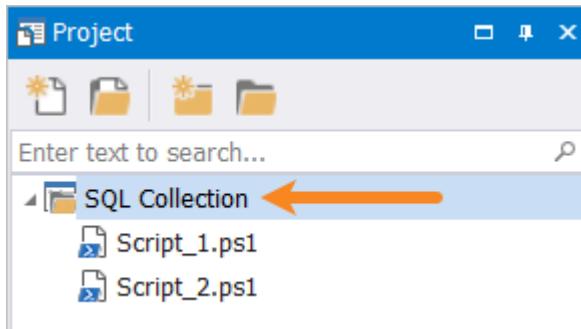


The collection project's Deployment properties are used to control the Packager, MSI, and Deployment behavior of the project as a whole, or as individual project files.

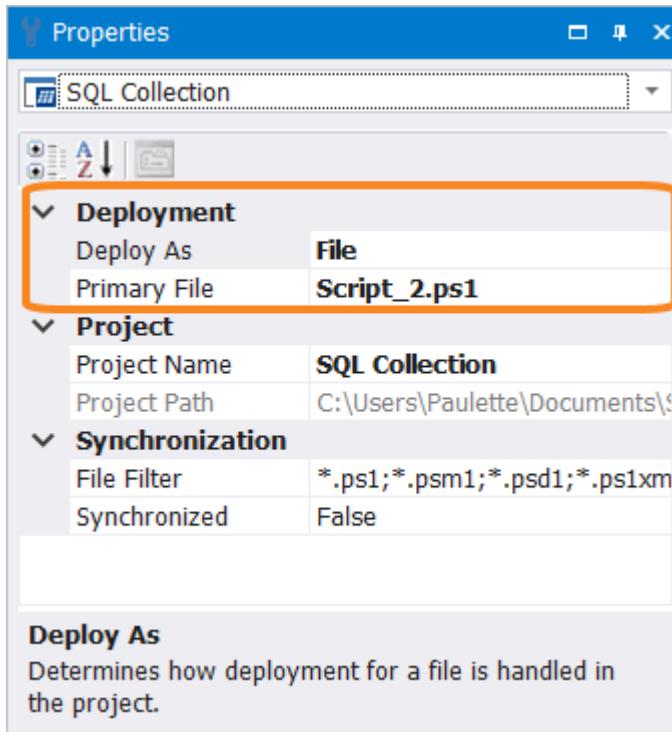
- A collection project's packaging is restricted because the files are all individual.

### To access the project's Deployment properties

Click on the collection project name in the [Project panel](#):



The project properties will display in the [Properties panel](#) [240]:



## Collection Project Deployment Properties

- **Deploy As**

Determines the deployment behavior of the Project as a whole. There are two **Deploy As** options:

File	<p>The project will handle each file individually from a deployment perspective. Each project file will maintain its own independent settings.</p> <p><b>When to use:</b></p> <p>Use this setting when you are using the Collection Project to group individual files that don't necessarily interact with each other. With this setting, you can package and deploy (publish) each script independently.</p>
Project	<p>The project will deploy all of the files as a whole. You must define a primary file for the purposes of the Packager and MSI builder.</p> <p><b>When to use:</b></p> <p>Use this setting when you have a group of files that interact and have a start/entry point script (i.e., a primary script that dot sources various secondary scripts). The primary script will get converted into an executable, and you can create an installer that includes the primary packaged script and all of the supporting files/scripts. The project files are also deployed (published) as a whole.</p>

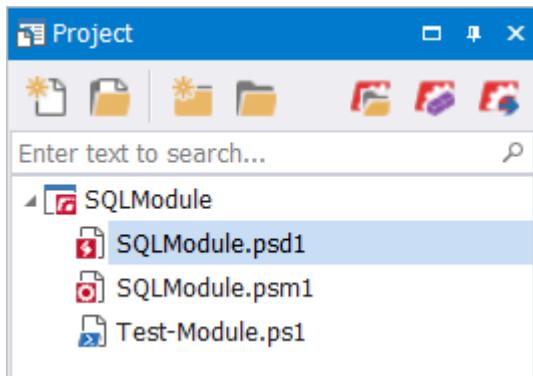
- Primary File

Designates the primary file for the project (*Deploy As = Project*). The primary file will be the file that is packaged into an executable, and all other files will be considered external content.

#### 9.1.4 Module Project

A script module is a library of PowerShell functions delivered together for some common purpose. Script modules are a good way to create packaged, reusable utilities that can be installed anywhere they are required.

The *module* project template includes everything that you need to get started:



The **psm1** file is where you define the functions that implement your module. A psm1 file is just a regular PowerShell script with a different extension. You add functions to the file in the same way that you would create a regular ps1 script file.

The **psd1** file is a module manifest file. It is used to provide extra metadata about a module including such things as:

- Module version number.
  - Author.
  - Description.
  - Prerequisites for executing the module: required PowerShell version, required CLR version, other modules and assemblies that must be present.
  - Export restrictions: lists of functions, variables, aliases and cmdlets to export.
-  Using a module manifest file allows you to cleanly separate your code from instructions and metadata about your code.
-  You can add multiple psd1 and psm1 files to a module project as long as they are located in a sub-directory and not in the root of the project folder.

The **Test-Module.ps1** script lets you test the functions and other features of your module.

## To use the **Test-Module.ps1** test script

---

- Import the module (be sure to import the correct version).
- Write commands that test the module features (you can include Pester tests).

## To run the **Test-Module.ps1** test script

---

- From any script in the project, on the **Home** tab click **Run**, or click **Run > Run in Console**.

**-OR-**

- Right-click on the **Test-Module.ps1** tab, then select **Run Script** or **Run Script in Console**.

**-OR-**

- In the [Project panel](#) , right-click on the **Test-Module.ps1** script, then select **Run Script File** or **Run Script in Console**.

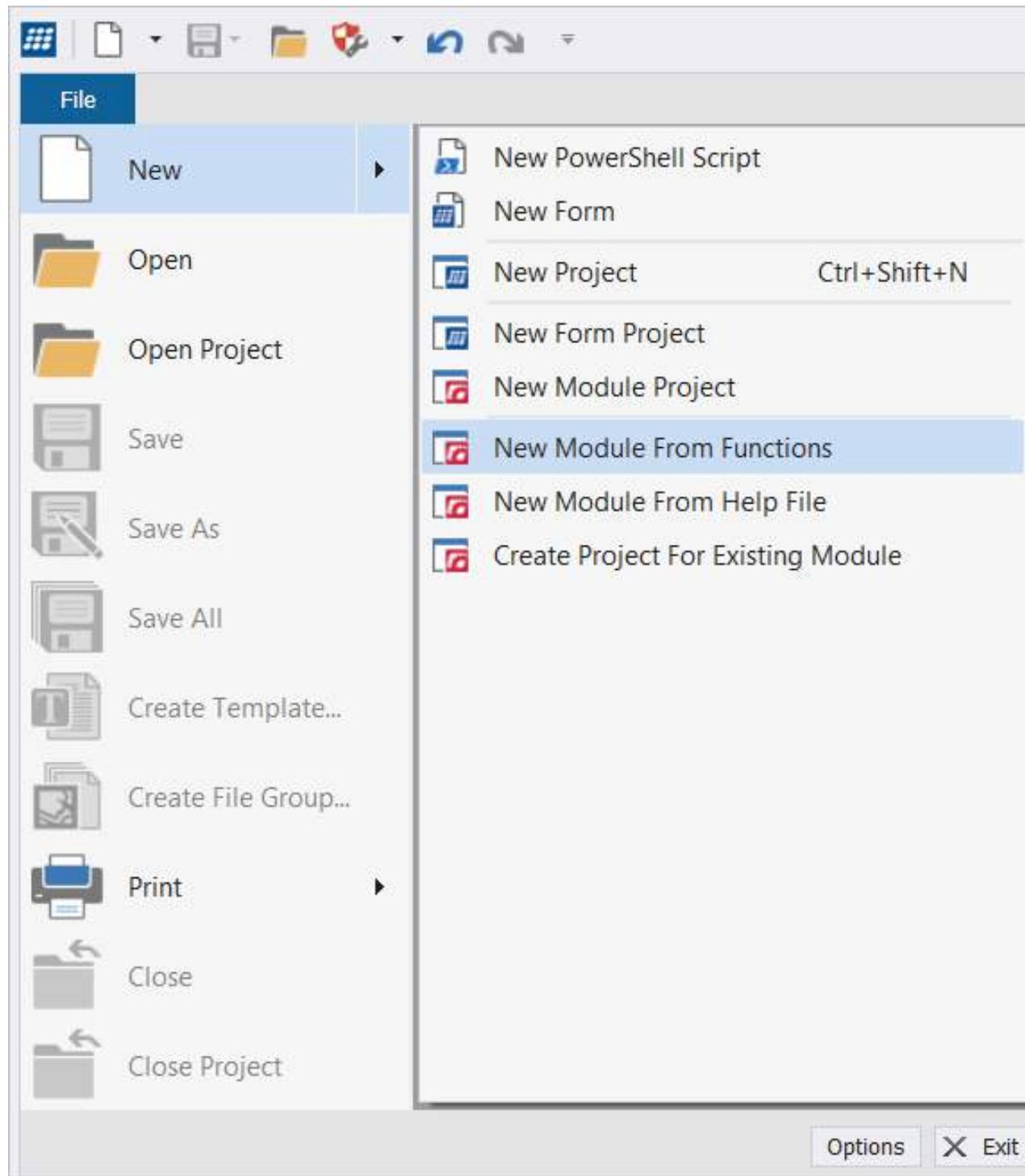
### 9.1.5 New Module from Functions

The **New Module From Functions** option allows you to import functions from various ps1 scripts and merge them into a new script module.

## How to create a New Module From Functions

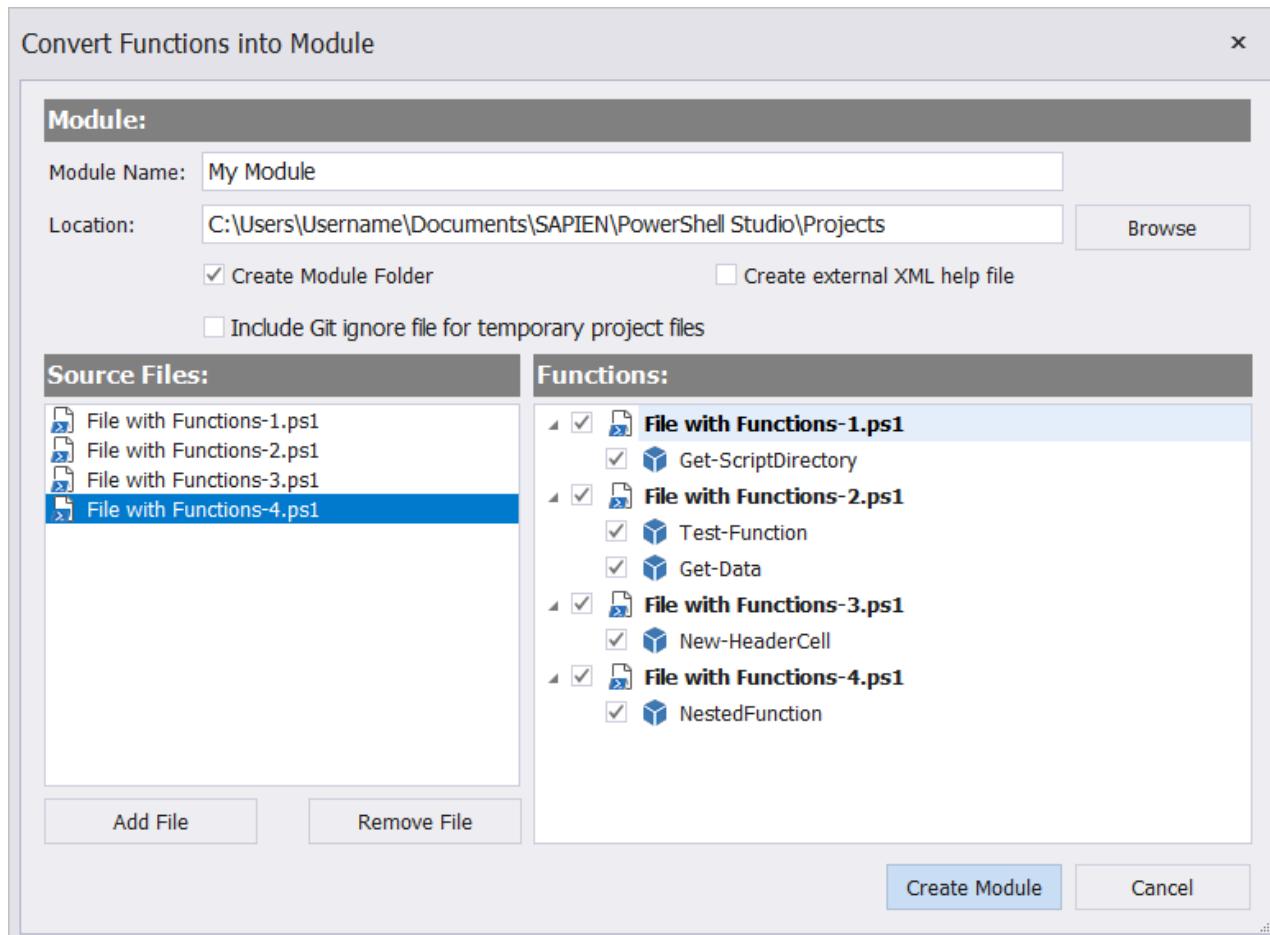
---

Select the **File** tab > **New** > **New Module From Functions**:



If a file is not already open, you will be prompted to select a file.

The Convert Functions into Module dialog allows you to select files and functions for the new module:



### Convert Functions into Module - Dialog Options

- **Module Name**

Enter the name you wish to give your new module.

- **Location**

Specify the folder where the module will be saved. The default is PowerShell Studio's project directory.

- **Create Module Folder**

This option creates a folder using the module's name and places all the generated files within that folder. This folder will be created in the folder specified by the Location field.

- **Create external XML help file**

This option will create an external XML help file for the module using the imported functions.

- **Include Git ignore file for temporary project files**

When checked, PowerShell Studio will create a .gitignore file for Git source control that filters any temporary project files.

- **Source Files**

Contains the list of files from which the selected functions will be extracted.

- **Add File**

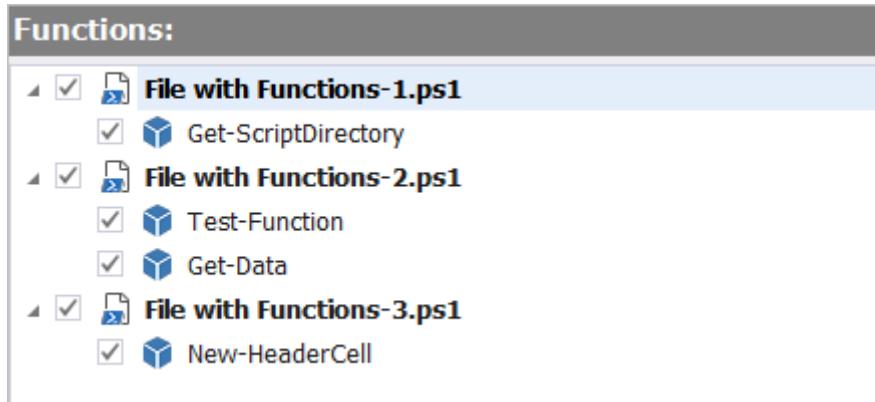
Use the *Add File* button to add ps1 scripts to the *Source Files*, in order to extract their functions.

- **Remove File**

Use the *Remove File* button to remove unnecessary files from the *Source Files* list. This can help de-clutter the functions list.

- **Functions**

The functions section contains a node for each file and a list of functions that are declared in each file. Select the functions to import into the new module:



➔ You can check and un-check all the functions in the file by checking/un-checking the file's node.

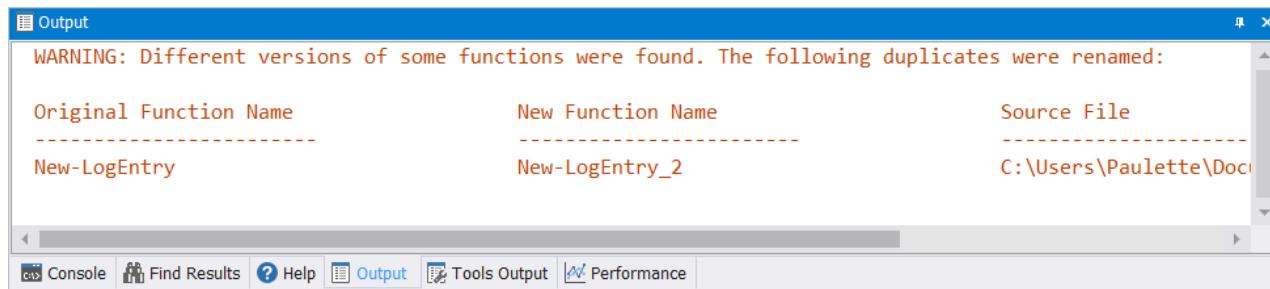
If a function references another function, it will have a *Referenced Functions* folder icon containing a list of all the referenced functions:



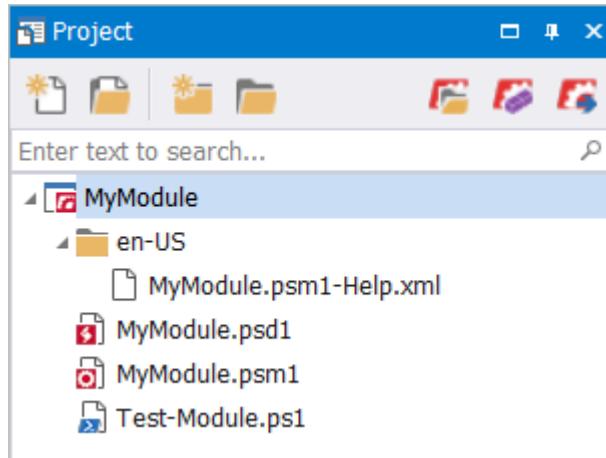
➔ When you check a function in the list that has references, it will automatically check all the referenced functions.

After you have selected the functions and configured the options in the Convert Functions into Module dialog, click **Create Module** to generate the new module.

ⓘ In some instances you may have a duplicate function that is defined in multiple files. PowerShell Studio will compare these functions, and if they are identical it will only insert the function once. If the functions are different, PowerShell Studio will automatically rename the duplicate. A warning will be displayed in the Output panel when this occurs:



The New Module From Functions project contains four files:



- <*ModuleName*>.psd1

The manifest file for your module.

- <*ModuleName*>.psm1

A PowerShell script containing all of the imported functions.

- <*ModuleName*>.psm1-Help.xml

The PowerShell XML Help file for the module, generated using the imported functions.

- Test-Module.ps1

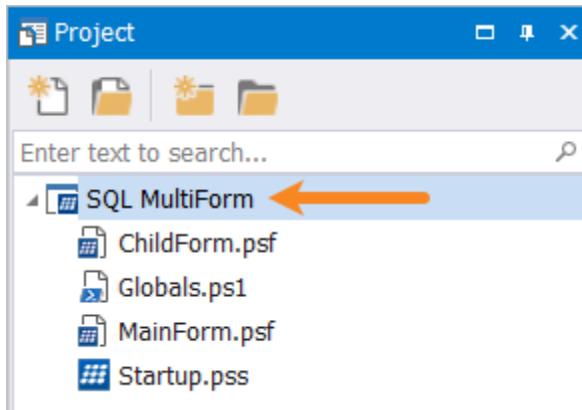
The PowerShell script code for testing your module.

## 9.2 Project Properties

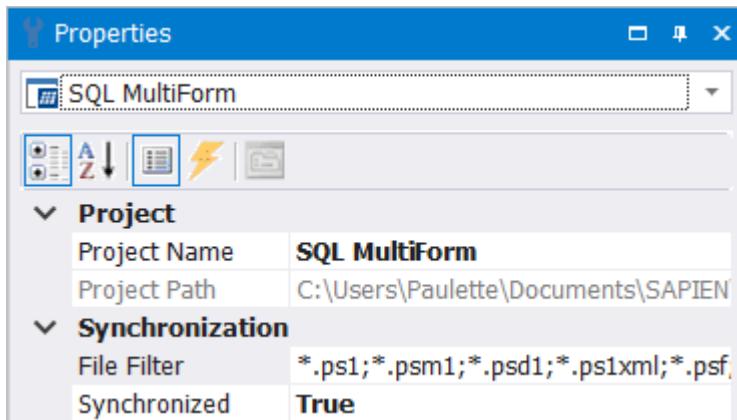
Project properties are shown in the Properties panel.

### How to view project properties

Click on the project name in the [Project panel](#) .



The project properties will display in the [Properties panel](#).



## Project Level Properties

- **Project**

- **Project Name**

Allows you to change the project name.

- **Project Path**

The location where the project is stored. This is not an editable property.

- **Synchronization**

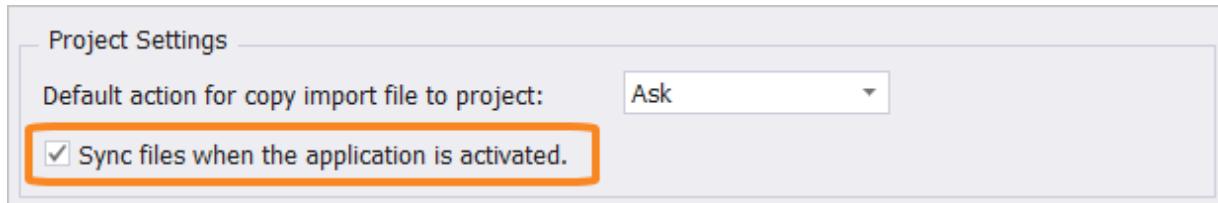
- **File Filter**

The file filter used to synchronize the project files.

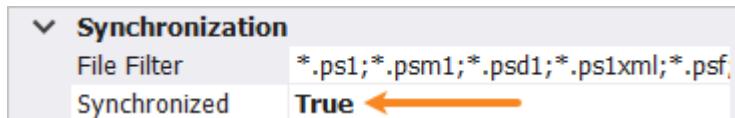
- **Synchronized**

Synchronize projects and files when the project is loaded.

This option allows you to trigger project file synchronization when the application regains focus (activated). In order for this feature to work, you must select **Sync files when the application is activated** in Options > General > Project Settings:



You also need to have a project open with file synchronization enabled (`Synchronized = True`):

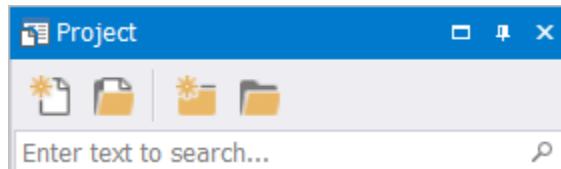


*Project sync on activate* ensures that the application can detect changes when you are making modifications to the project's folder structure outside of PowerShell Studio.

## 9.3 Managing Project Files

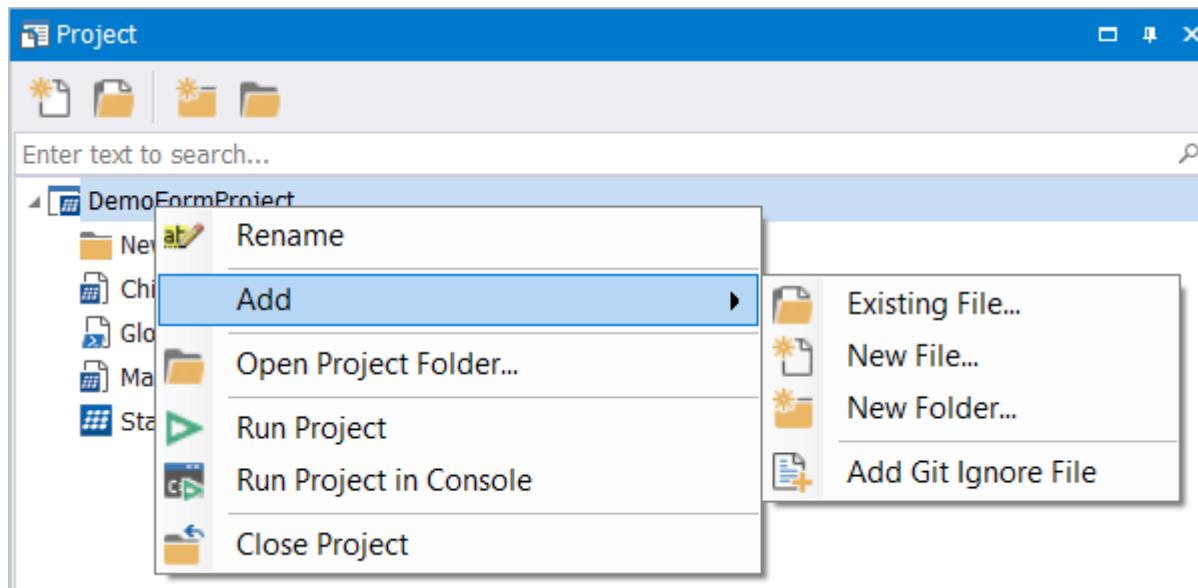
Project files and folders are managed in the Project panel.

The top of the [Project panel](#) has buttons for common project tasks, as well as a search box:



### To access the project level options

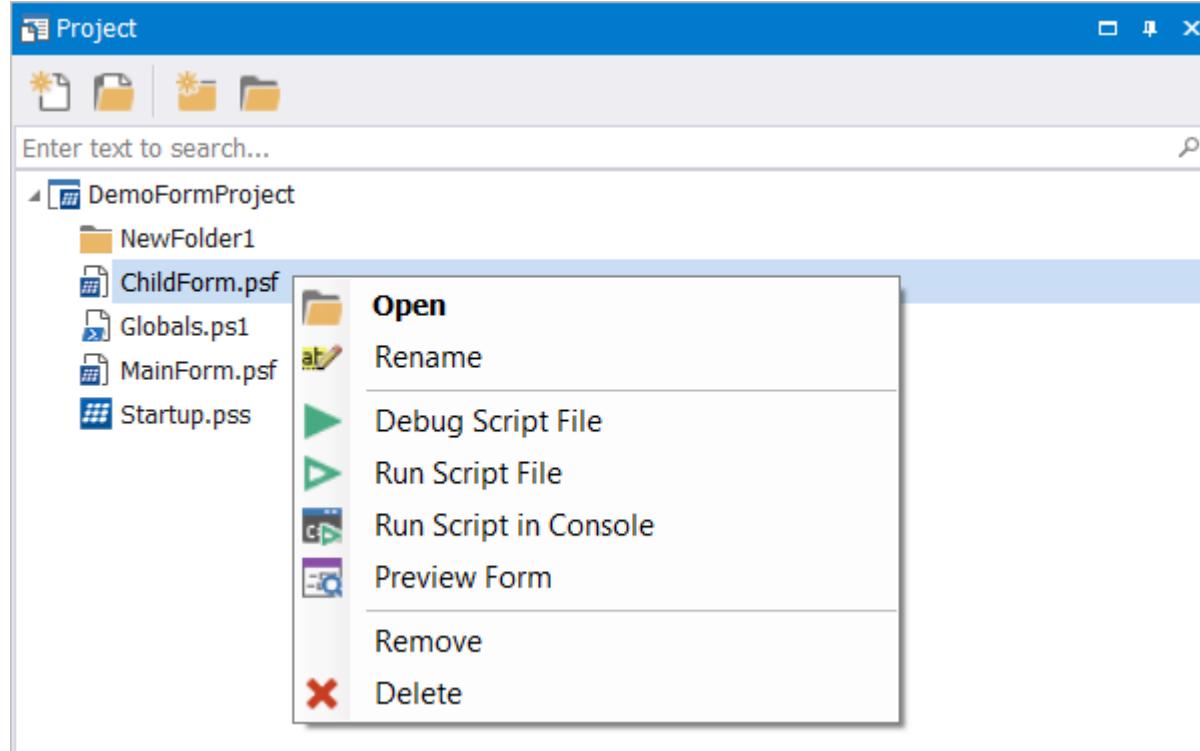
Right-click on the project name:



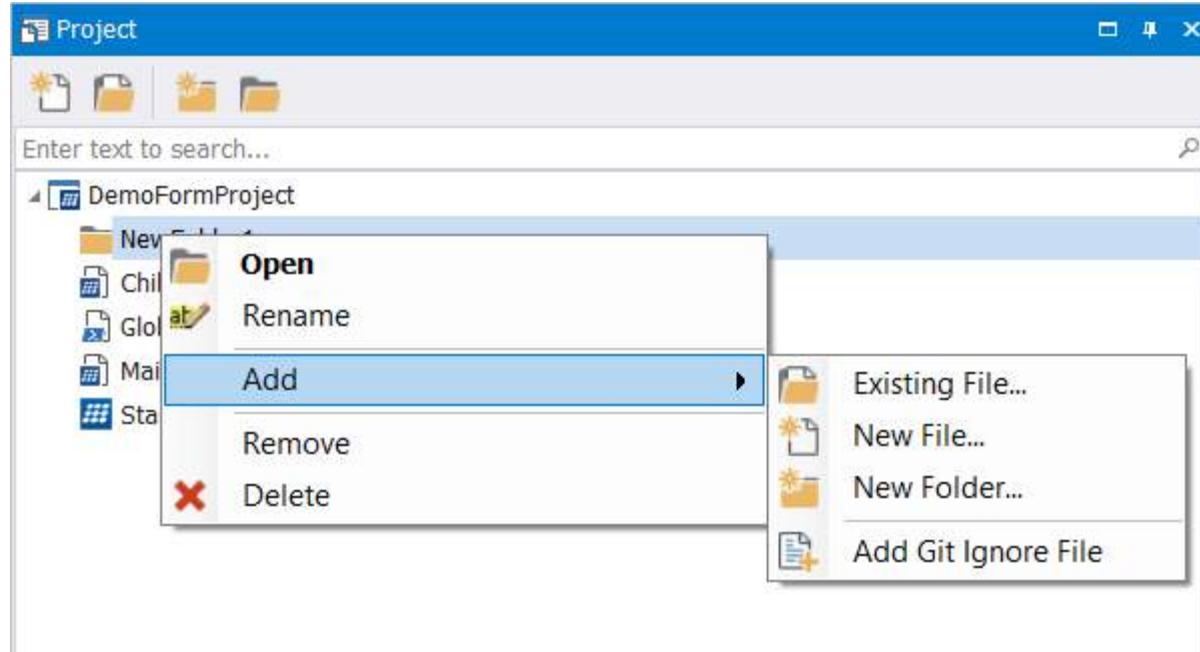
## To access context sensitive options

Right-click on a project file or folder:

### *Project file options*



### *Project folder options*

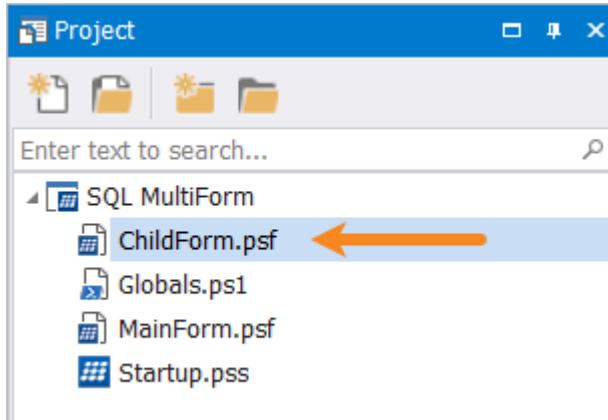


## 9.4 Project File Properties

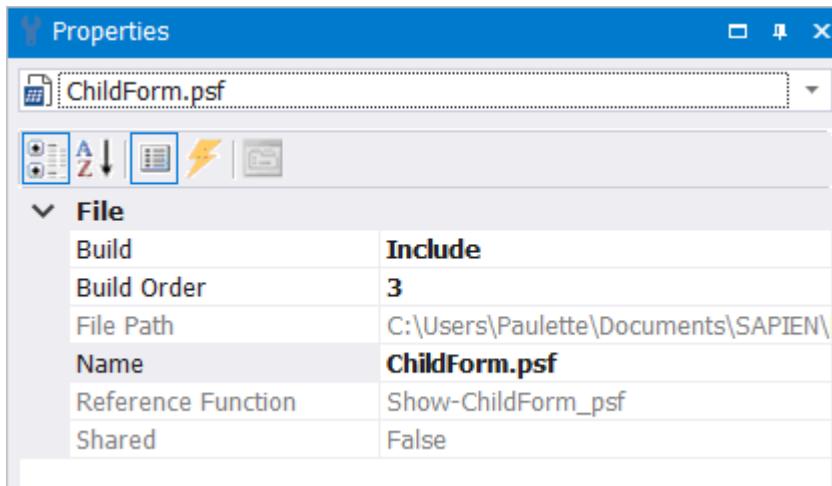
A project's file properties are shown in the Properties panel.

### How to access the project file properties

Click on the project name in the [Project panel](#):



The file properties will display in the [Properties panel](#):



### Project File Properties

- **Build**

This property determines what PowerShell Studio does with a file when you deploy / export a project. Three options are supported:

- **Include**

The file is included in the build. The Reference Function properties are used to help integrate the file contents into the shell.

- **Exclude**

The file will not be included in the build.

- **Content**

The file will be included in the build but any code contained in the file will not be integrated into the shell. This is a useful option when you want to include data files in your project.

- **Build Order**

Sets the order in which the file is built / merged by the project.

- **File Path**

The location where the file is stored. This is not an editable property.

- **Name**

The name of the file can be edited here.

- **Reference Function**

The name of the function that invokes the project file.

- **Shared**

If enabled, the functions and variables declared in the ps1 file can be referenced by other project files, and you will not be able to invoke the file by its reference function.

- **Export Function (*Module projects only*)**

Exports the functions defined in the file. Requires the project's Auto Export Functions property to be *True*.

## Invoking Project Files

The Reference Function property described above references the function that allows simple invocation of a project file. If you add a script file called *Utilities.ps1* to a project and you examine its properties, you will see that PowerShell Studio has generated a Reference Function called *Invoke-Utilities\_ps1*. You can use this name elsewhere in your project scripts to run the code in *Utilities.ps1*.

For *Form* projects, PowerShell Studio uses this method to load the first form in a project from the project startup script (Startup.pss):

```
27 if((Show-MainForm_psf) -eq 'OK')  
28 {  
29  
30 }
```

## 9.5 Adding Script Parameters to Projects

Adding a *Param* block to the Startup.pss file allows you to provide startup parameters to your project when it runs.

The runtime behavior depends on how the project is started:

- If you start your project from the **console**, then you provide parameter values on the command line, separated by spaces.
- If you are starting a **Forms** project, then PowerShell Studio will prompt you for the parameter values.
- If you **package** a project into an **EXE** file, then you must always provide startup parameters on the command line.

You can also add Param blocks to forms and scripts in your project, as long as their Shared property is not set to true. Simply add a Param block to the beginning of the file and provide parameter values when you invoke the file.

For example, if we wanted to pass the current user's name to each form in our script, we could add a Param block to each form:

```
1 Param (
2     [string]$username
3 )
4
5 $form1_Load={
6     #TODO: Initialize Form Controls here
7     $MainForm.Text = $username
8 }
9
```

Then supply an appropriate value when we call the form:

```
27 if((Show-MainForm_psf $env:USERNAME) -eq 'OK')
28 {
29 }
30 }
```

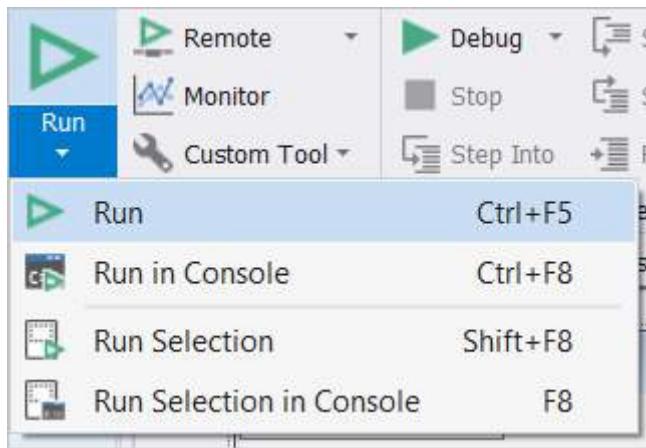


## 9.6 Running a Project

Options to run a project are available on the [Home ribbon](#)  138, and also in the [Project panel](#)  225.

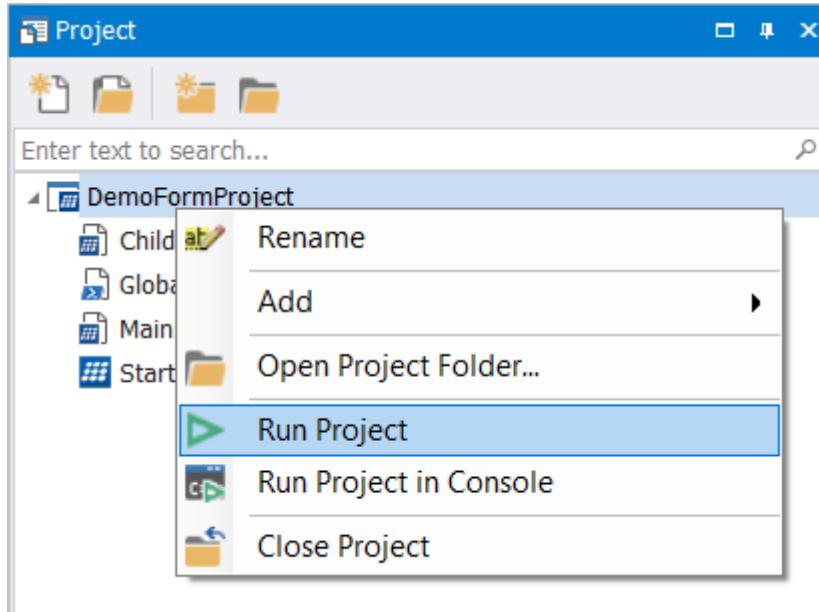
### To run a project

- Click the Home tab > Run menu > Run (**Ctrl+F5**):



-OR-

- Right-click on the project name in the Project panel and choose one of the Run options:



- The execution options available will depend on the project type.
- You can also run each file individually by right-clicking on a file in the [Project panel](#).

## 9.7 Exporting a Project

Exporting a project converts the project into a single script file that contains all of your code, plus the auto-generated code produced by PowerShell Studio. You can export a project to a file, or to the clipboard.

The export options are located on the Deploy tab > Export section:



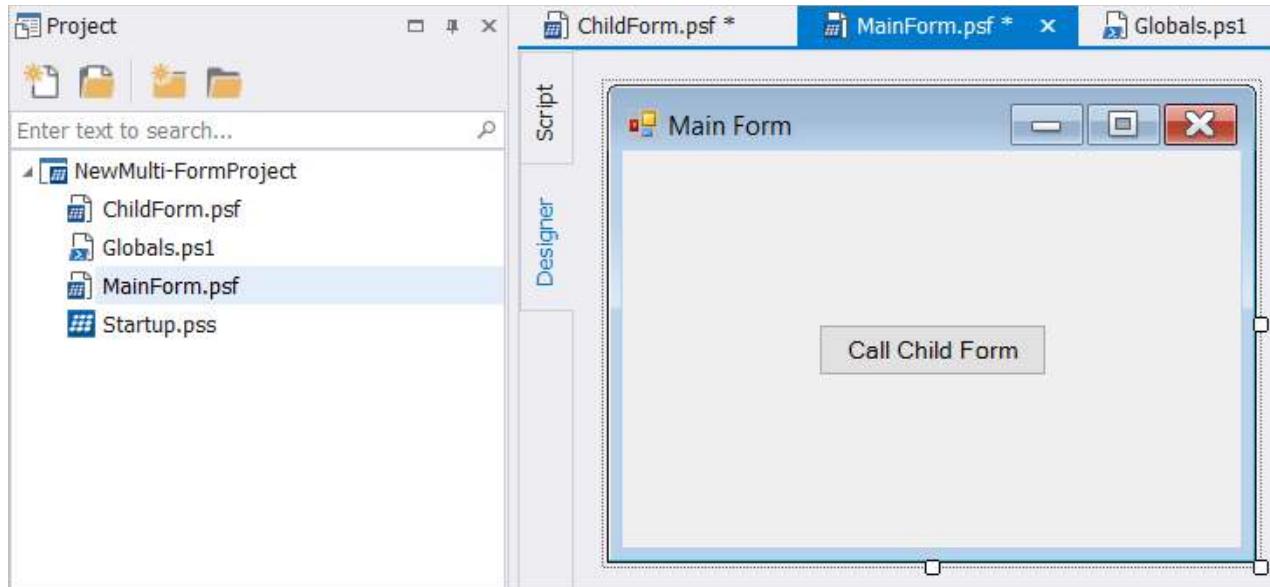
### Project Export - Options

- **Export to File**  
Exports the project to a ps1 file.
- **Export to Clipboard**  
Copies the project to the clipboard.

## 9.8 Form Return Variables

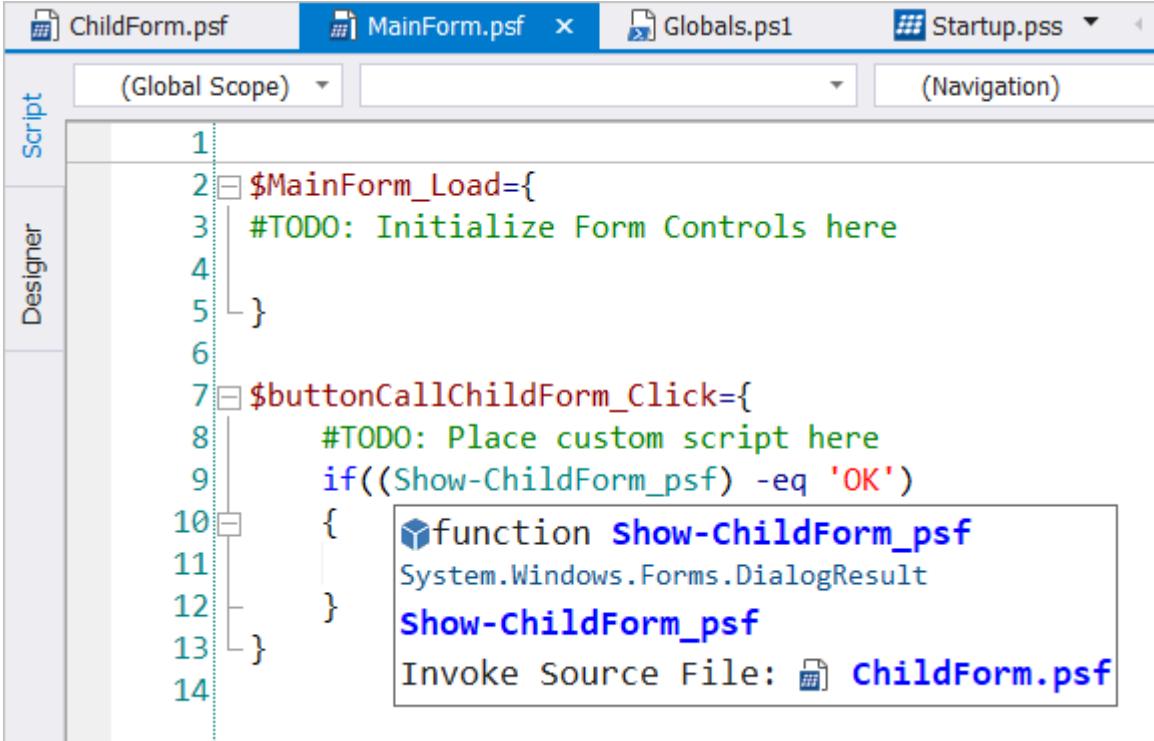
In order to simplify working with forms in a project, PowerShell Studio will create special variables that allow you to refer directly to a property of controls on a child form. These variables are accessible from the script that invokes the form's reference function (see [Invoking Project Files](#) [281]).

To demonstrate this, create a new *Multi-Form* project (**File > New > New Form Project > Multi-Form Project**):



The Multi-Form project template includes a button on the main form called 'Call Child Form'.

Click on the main form Script tab to view the code. Notice that the CallChildForm button click handler includes the Show-ChildForm\_psf reference function to call the child form:



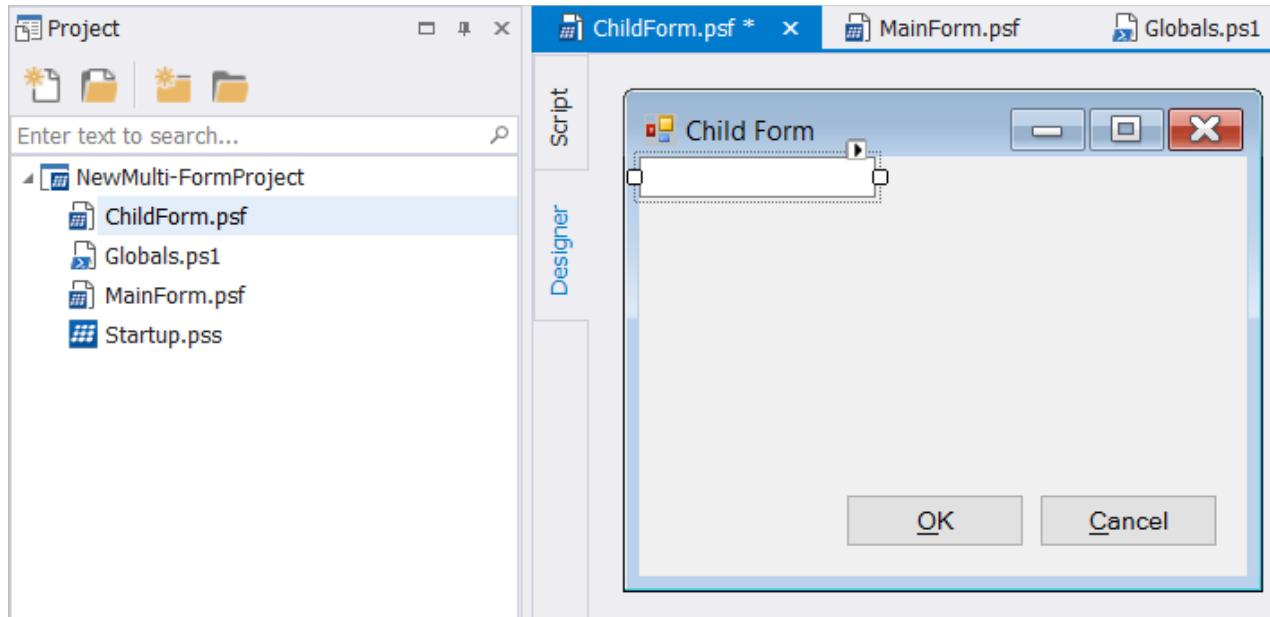
The screenshot shows the PowerShell Studio interface with the 'Script' tab selected. The code editor displays the following PowerShell script:

```
1
2 $MainForm_Load={
3     #TODO: Initialize Form Controls here
4
5 }
6
7 $buttonCallChildForm_Click={
8     #TODO: Place custom script here
9     if((Show-ChildForm_psf) -eq 'OK')
10    {
11        function Show-ChildForm_psf
12            System.Windows.Forms.DialogResult
13        Show-ChildForm_psf
14    }
15 }
```

A tooltip box is overlaid on the code, highlighting the function definition for `Show-ChildForm_psf`. The tooltip contains the following text:

function Show-ChildForm\_psf  
System.Windows.Forms.DialogResult  
**Show-ChildForm\_psf**  
Invoke Source File: [ChildForm.psf](#)

Next, add a textbox to the child form (Child Form Designer tab > Toolbox panel > Textbox control):



 For information about Windows Forms controls, see [Panels > Toolbox Panel](#) 

PowerShell Studio will make the text property of the textbox in the child form directly accessible in the main form through a variable called `$ChildForm_textbox1`. The screenshot below shows PowerShell Studio's Intellisense suggestion when you type '`$chi`' in the main form:

The screenshot shows the PowerShell Studio interface with the 'Script' tab selected. In the main editor area, there is a script for 'buttonCallChildForm\_Click'. A tooltip is displayed over the variable '\$ChildForm\_textbox1', which is highlighted in the code. The tooltip provides information about the variable:

- variable \$ChildForm\_textbox1
- System.String
- Represents text as a series of Unicode characters.
- [View Type](#)
- [MSDN Help](#)

The code in the editor is as follows:

```
1
2 $MainForm_Load={
3     #TODO: Initialize Form Controls here
4
5 }
6
7 $buttonCallChildForm_Click={
8     #TODO: Place custom script here
9     if ((Show-ChildForm_psf) -eq 'OK')
10    {
11        $chi
12    }
13 }
14
```

The cursor is positioned at the start of the variable name '\$chi' in line 11.

This example illustrates that you can access a child form variable in the the main form code.

These form return variables make it easy to gather the results from data entry forms.

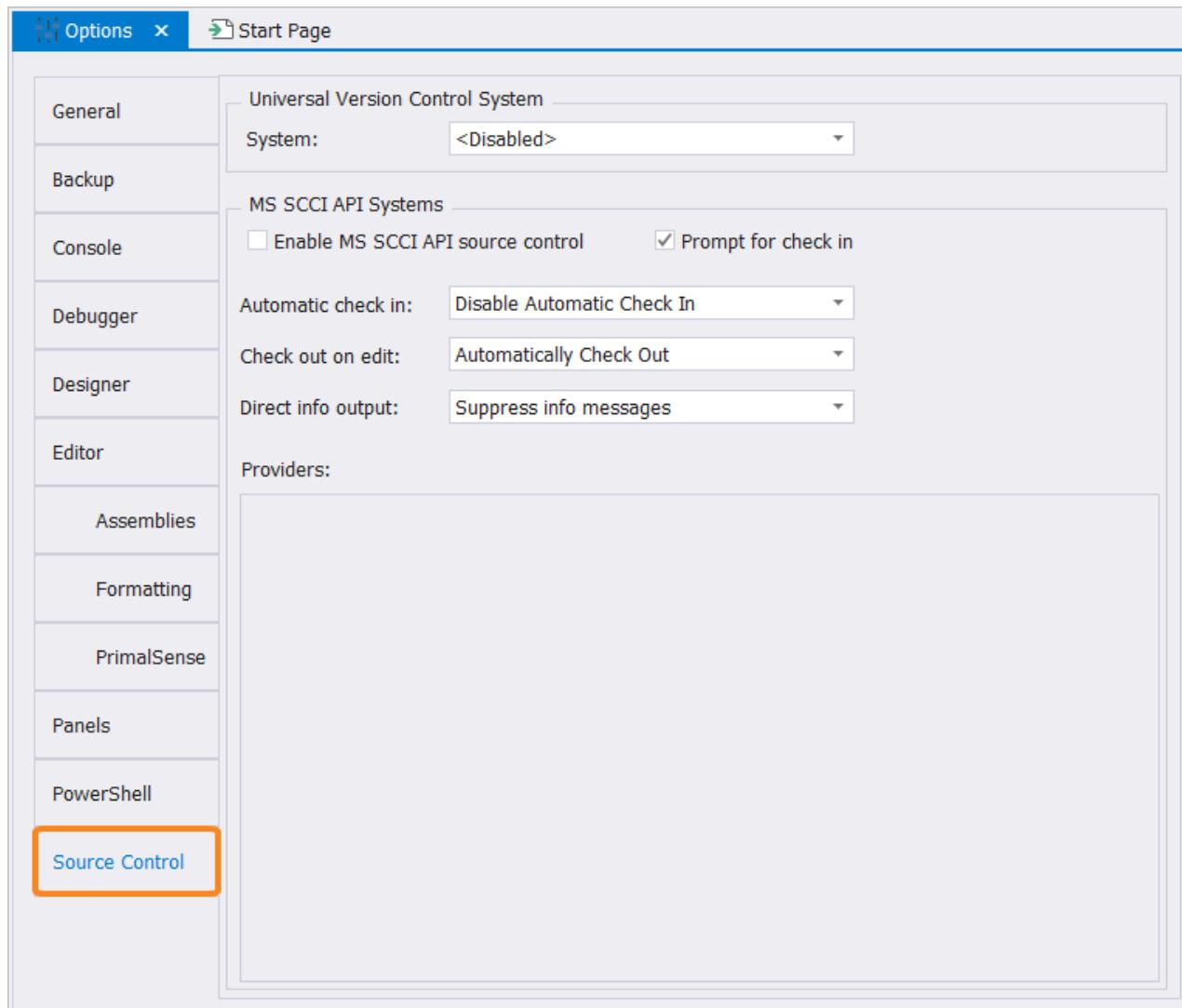
The controls that support this mechanism are summarized below, along with the type of data that they return:

<i>Control</i>	<i>Return Type Property and Data Type</i>
Checkbox	Checked (Boolean)
CheckedListBox	Selected item (string)
ComboBox	Selected item (string)
DataGridView	SelectedCells (DataGridViewCellCollection)
DateTimePicker	Selected date (DateTime)
ListBox	Collection of selected items(string)
ListView	Collection of selected items(string)
MonthCalendar	Selected date (DateTime)
NumericUpDown	Selected value (Decimal)
RadioButton	Checked (Boolean)
RichTextBox	Text (string)
Textbox	Text (string)
Tracker	Value (int)
Treeview	Selected Node (string)

## 9.9 Projects and Source Control

Projects can be managed as a unit through PowerShell Studio's source control integration.

Before managing a project through source control, [source control integration must first be configured](#) [302].



## To work with source control

In the [Project panel](#), right-click on the project name:

- **Add to source control**

Adds the project to source control.

- **Check in**

Checks in one or more project files which have changed or not yet been checked in.

👉 Individual files within the project can be checked in or out independently, but checking files in together as a project helps to simplify file and source control management.

## 10 Packaging Scripts

PowerShell Studio contains the Script Packager™, which can package single or multiple scripts, supporting files, and COM components into a single, standalone executable file (.exe).

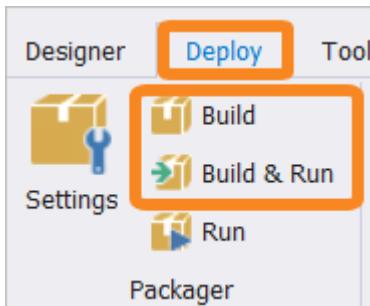
### 10.1 Creating a Script Package

This topic shows you how to create a script package.

-  If this is your first package, begin by [setting up the Script Packager](#).

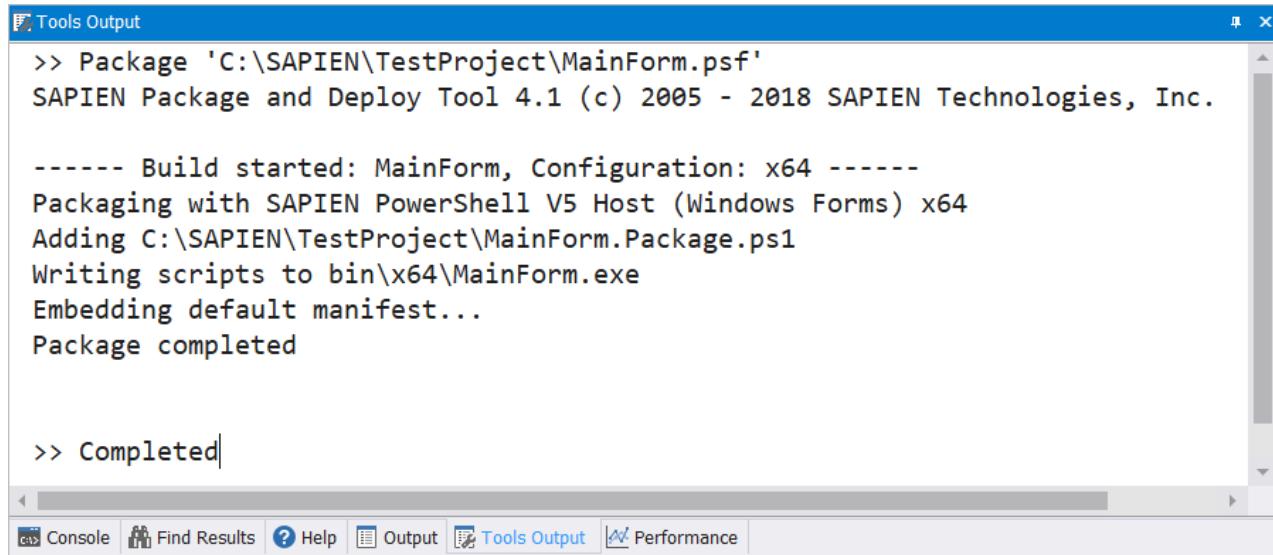
#### To create a package

- Click **Deploy** on the ribbon, then in the Packager section click **Build** or **Build & Run**:
  - Build (Ctrl+F7)** - Creates an executable file from the active document
  - Build & Run** - Creates an executable file from the active document and executes it.



PowerShell Studio checks the syntax of the designated files and packages them into an executable file (.exe).

If your build is successful, information about the new executable file is displayed in the Tools Output panel:



```
>> Package 'C:\SAPIEN\TestProject\MainForm.psf'
SAPIEN Package and Deploy Tool 4.1 (c) 2005 - 2018 SAPIEN Technologies, Inc.

----- Build started: MainForm, Configuration: x64 -----
Packaging with SAPIEN PowerShell V5 Host (Windows Forms) x64
Adding C:\SAPIEN\TestProject\MainForm.Package.ps1
Writing scripts to bin\x64\MainForm.exe
Embedding default manifest...
Package completed

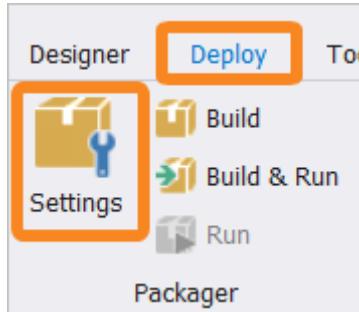
>> Completed
```

## 10.2 Setting up the Script Packager

The Script Packager contains everything you need to customize your executable files and create a package.

### To open Packager Settings and configure a script package

1. Click **Deploy** on the ribbon, then click **Settings** in the Packager section to open the Script Packager interface:



2. Select the desired settings in the Script Packager interface (see details below):

- [Script Engine](#) [290]
- [Output Settings](#) [292]
- [Execution Restrictions](#) [298]
- [Version Information](#) [299]
- [Build Options](#) [300]

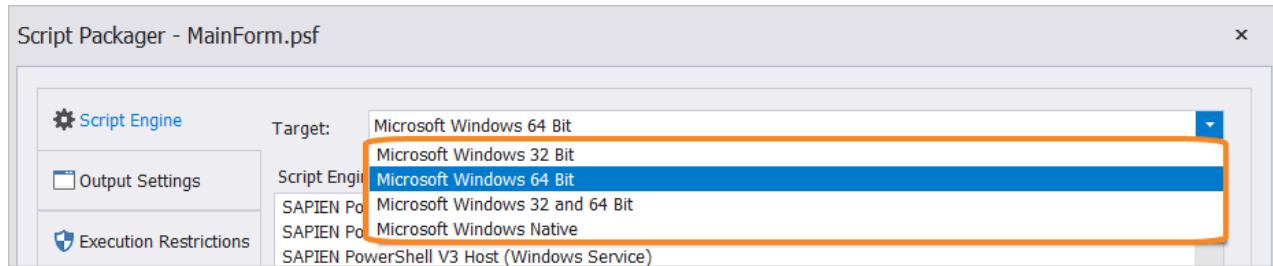
## Script Engine

### Target Platform

The Script Packager provides four options for building executables:

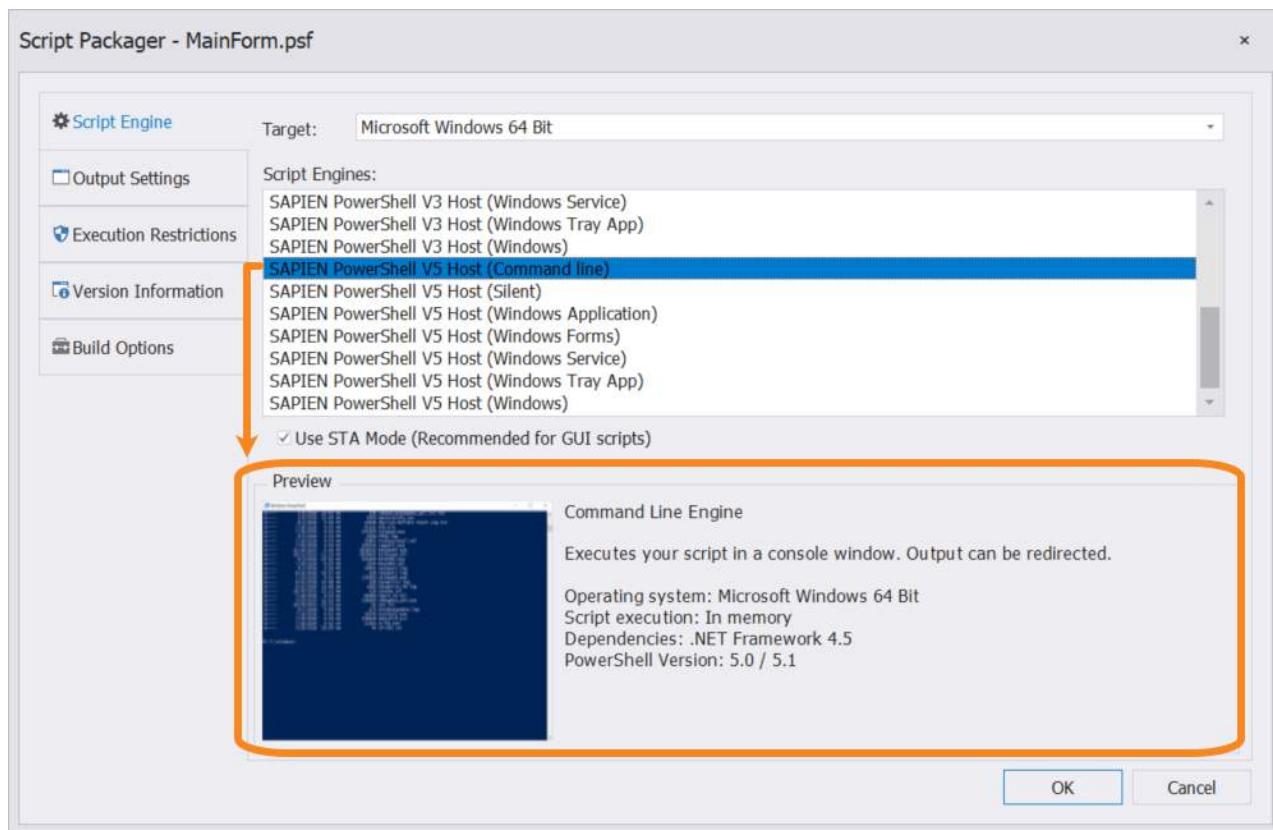
- **Microsoft Windows 32 Bit** will generate a 32 bit executable.
- **Microsoft Windows 64 Bit** will generate a 64 bit executable.
- **Microsoft Windows 32 and 64 Bit** will generate a 32 bit and a 64 bit executable.
- **Microsoft Windows Native** will create a starter executable which will launch the correct version depending on the current platform.

Select the desired platform from the options in the Target drop-down list:



## Script Engines

Each script engine option provides a preview of what the selection will do:



- i** STA (Single Threaded Apartment) Mode allows you to start your script in single threaded mode. This is essential when your script uses forms to interact with the Windows GUI. Some GUI controls require STA mode in order for them to function correctly.

## Output Settings

---

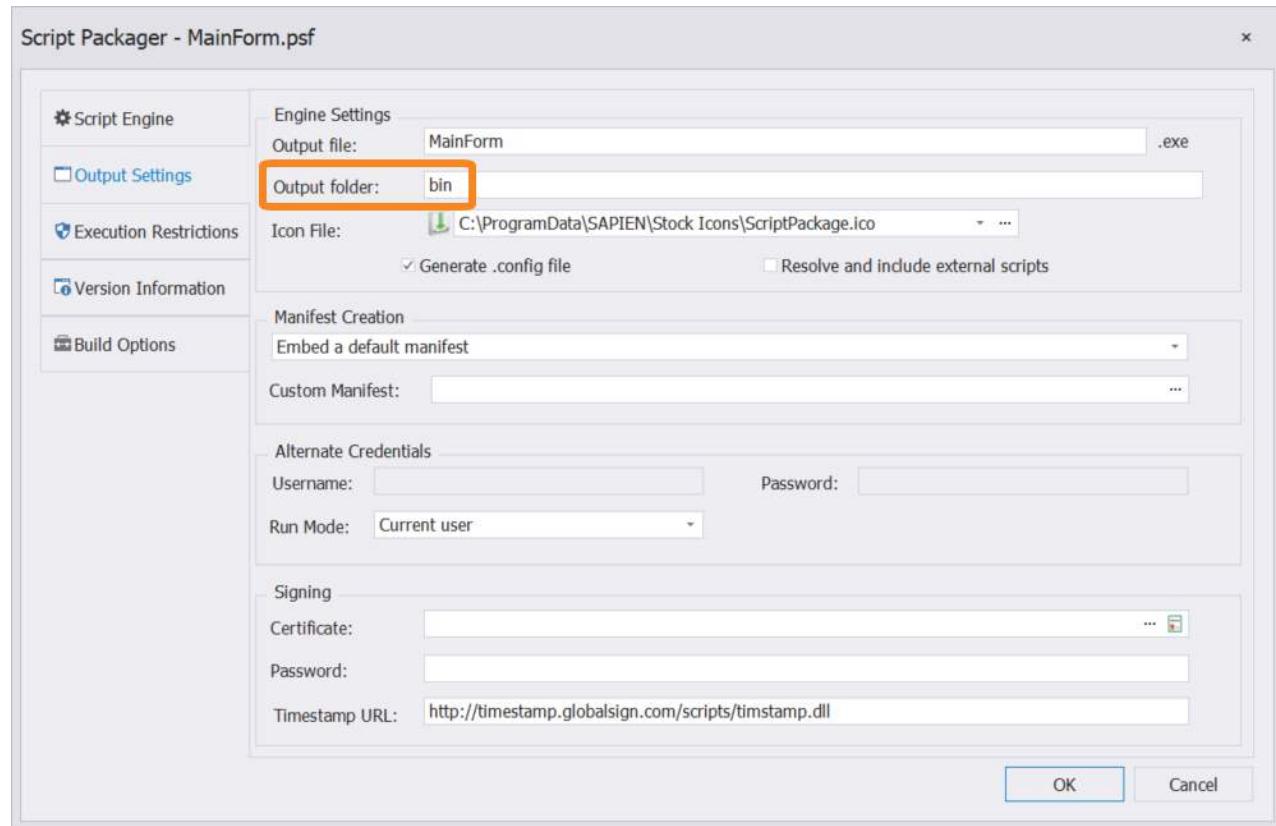
### Output Settings Options

---

<b>Output file</b>	Filename of the executable.
<b>Output folder</b>	Folder for the executable.
<b>Icon file (optional)</b>	A custom icon (.ico) for the executable.
<b>Generate .config file</b>	Generates a .config file.  If you select <i>Windows Native</i> , .config files will be generated for all three of the .exe files.
<b>Resolve and include external scripts</b>	The code of external scripts will get injected into the packaged script when building the executable.  Enable this option to resolve dot sourced files while packaging.
<b>Manifest creation</b>	Options for the manifest file, including a custom manifest.  (This is an executable manifest, not a Windows PowerShell module manifest.)
<b>Custom manifest</b>	Opens a file to the specified line.
<b>Alternate credentials</b>	Uses the credentials of the specified user to run the scripts in the executable file.
<b>Run mode</b>	<b>Current user:</b> Runs scripts with the permissions of the user who runs the executable file.  <b>RunAs:</b> Runs scripts with the permissions of the specified user in the specified user's environment.  <b>Impersonate user:</b> Switches to the security context of the specified user, but uses the environment (e.g. network profiles, mapped drives, environment variables) of the current user.
<b>Signing</b>	Specify the code signing certificate to sign your executable. If you specify a PFX file that requires a password, include it here.  The Timestamp URL creates a timestamp for the signature used to sign the file, allowing the signature to remain valid even after the certificate expires.

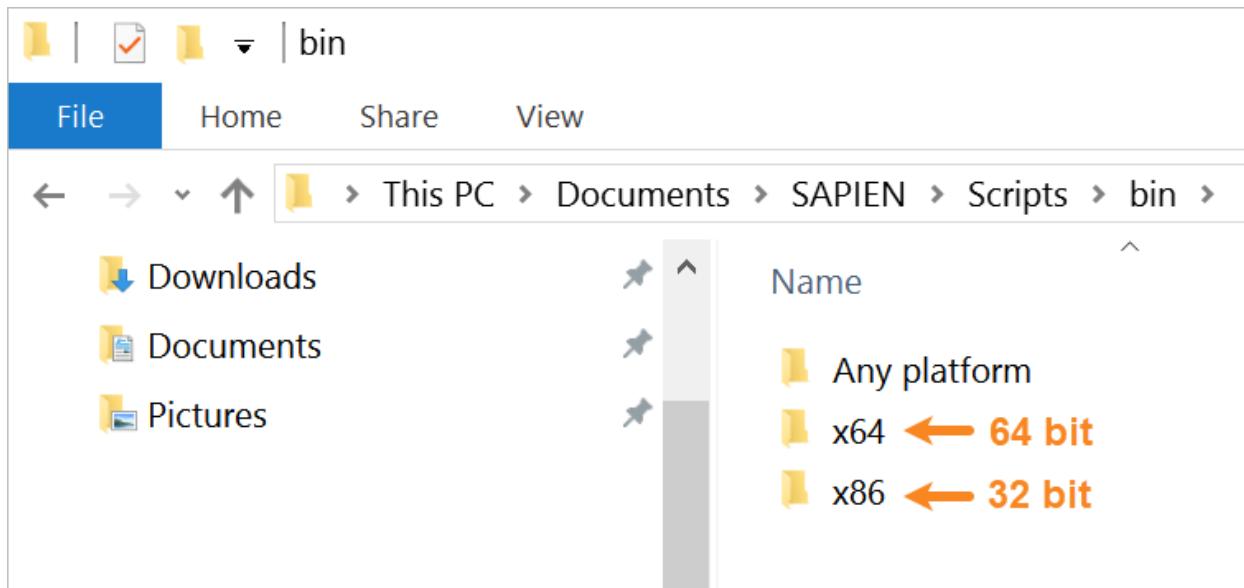
## Engine Settings

The packaged executable files are generated in a platform specific folder under a common folder. It is recommended that you leave the common folder default name of bin for consistency:

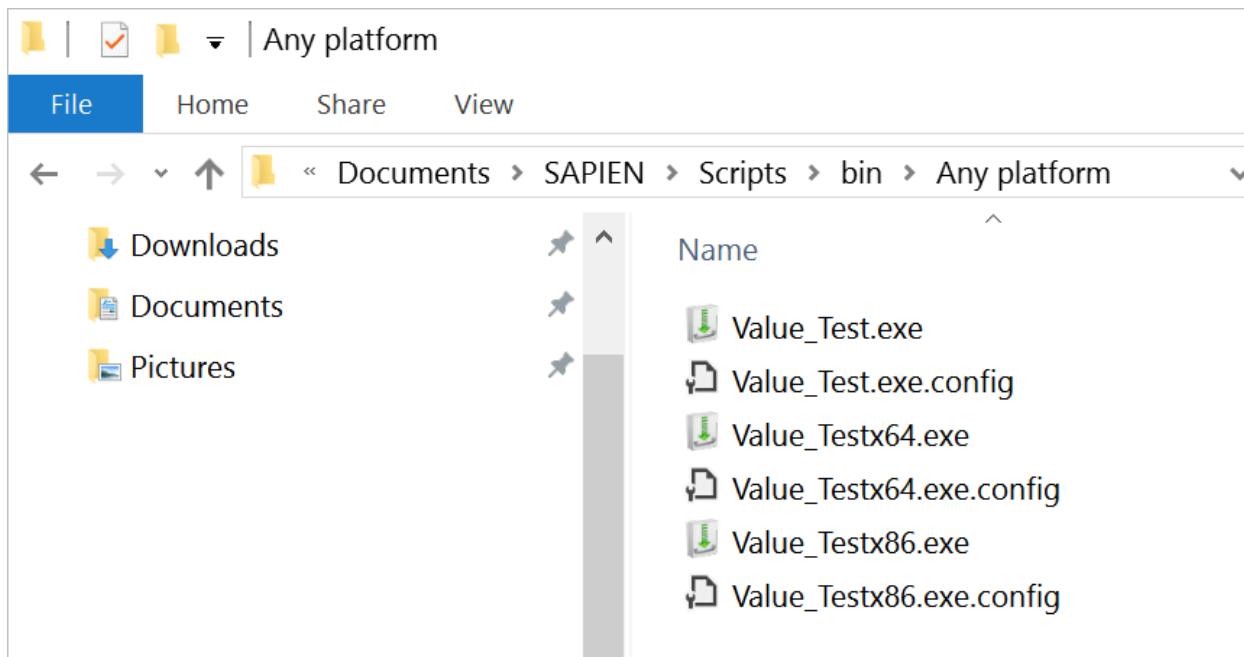


The build target you select will determine the platform specific folder that the packaged file(s) are generated in:

- 32 bit files will be in ***bin\x86***
- 64 bit files will be in ***bin\x64***
- 32 bit and 64 bit files will be in their respective folders (***bin\x86*** and ***bin\x64***)



- Windows Native executables will be in *bin\Any platform*



Choosing the *Windows Native* option will generate three .exe files:

- <app>x86.exe and <app>x64.exe are your actual packaged script.
- <app>.exe is a starter application that will execute the right package for the current platform.

You must install or deploy all three files together for your application to work. The starter application will receive the same icon, digital signature, and manifest as the packaged files, so a shortcut to <app>.exe will create the same experience.

-  If you select both *Windows Native* and *Generate .config file*, then .config files will be generated for all three of the .exe files.

### External Scripts

Select **Resolve and include external scripts** to deploy dot sourced files with the executable. If this option is enabled, the code of the external scripts will get injected into the packaged script when building the executable.

- Files specified with or without single and double quotes are supported. Files that do not exist will issue a warning. If you have a dot source statement inside a comment block, the file will be inserted into the comment block.
- Using a line comment will prevent a file from being resolved.
- If you need to resolve only some but not all external files, you can use a different case for the file extension:
  - ./include/lib.ps1 will be resolved by the packager.
  - ./include/lib.PS1 will *not* be resolved.

In other words, the statement is case sensitive; the actual filename's case is not relevant.

```
28 #>
29
30 # ."/include/oldhello.ps1"
31
32 #. "./oldinclude/someotherfile.ps1"
33
34 #./includes/hello.ps1"
35
36 Write-Host "Yes, it works"
37
38 .$PSScriptRoot\includes\functions.ps1
39
40 get-platform
41
42 $args
43
```

### Alternate Credentials

By default, the scripts in a package run in the security context of the user who runs the package. You can specify alternate credentials (a username and password) that will be used to run the scripts.

The screenshot shows a 'Alternate Credentials' dialog box. It contains three main fields: 'Username' (with a placeholder 'Enter a value...'), 'Password' (with a placeholder 'Enter a value...'), and a dropdown menu labeled 'Run Mode' with the option 'Current user' selected.

The alternate credentials you supply must be available (either as local or domain accounts) on any computer where the packaged executable will run. Also, the credentials must generally have local administrator privileges on the computer where the package will run.

### ***Alternate Credentials options:***

- **Username**

Username of the specified user that will run the scripts in the package.

**i** To specify a domain, use `username@domainname` format, not `domain\user` format. Do not specify a domain or computer name for local accounts.

- **Password**

Password of the specified user that will run the scripts in the package.

- **Run Mode**

Select the user profile that will run the scripts in the package.

- **Current user**

Runs scripts with the security context of the current user, in the current user's environment.

- **Impersonate user**

Runs scripts with the security context of the specified user, in the current user's environment.

- **RunAs user**

Runs scripts with the security context of the specified user, in the specified user's environment

### **Elevate Regular User to Full Administrator**

This section explains how to package a script as an executable, with the objective of allowing a regular user to accomplish a task that requires full administrator privileges.

#### **Some background:**

Since Windows Vista, the Administrator security token is split—you cannot simply logon as Admin and do anything you need to do. An Admin must *elevate* in order to accomplish certain tasks (e.g., when accessing or modifying certain system areas). This has ramifications for packaging executables—you cannot successfully use a run mode of *RunAs* or *Impersonation*, and also *elevate* at the same time.

#### **When selecting *RunAs* or *Impersonation*:**

- The specified credentials are stored inside the packaged executable, encrypted.
- When the packaged executable is launched, it uses certain API calls to create a new security token (*Impersonation*) or run itself with the specified credentials (*RunAs*). The executable needs to load and execute in order for this to happen.

## When selecting a *manifest for elevation*:

- The manifest is embedded in the executable—unencrypted—because Windows needs to read this information.
- Windows will load and evaluate this manifest **before** any code is executed. If you run this from a regular user, you will be prompted for Admin credentials and also to verify elevation. The credentials stored inside the package have no effect at this point because they would only be applied after the fact.

Essentially, due to the way Windows evaluates manifests, elevation happens **before RunAs / Impersonation**—but it needs to be the other way around to avoid prompts and to not give regular users Admin privileges. The Script Packager accomplishes this via a two-step process:

1. Starter.exe—a simple script packaged as an executable that includes; the Admin credentials, a run mode of either *RunAs* or *Impersonation*, and instructions to launch your script.
2. Your script—packaged as an executable, with a manifest for elevation.

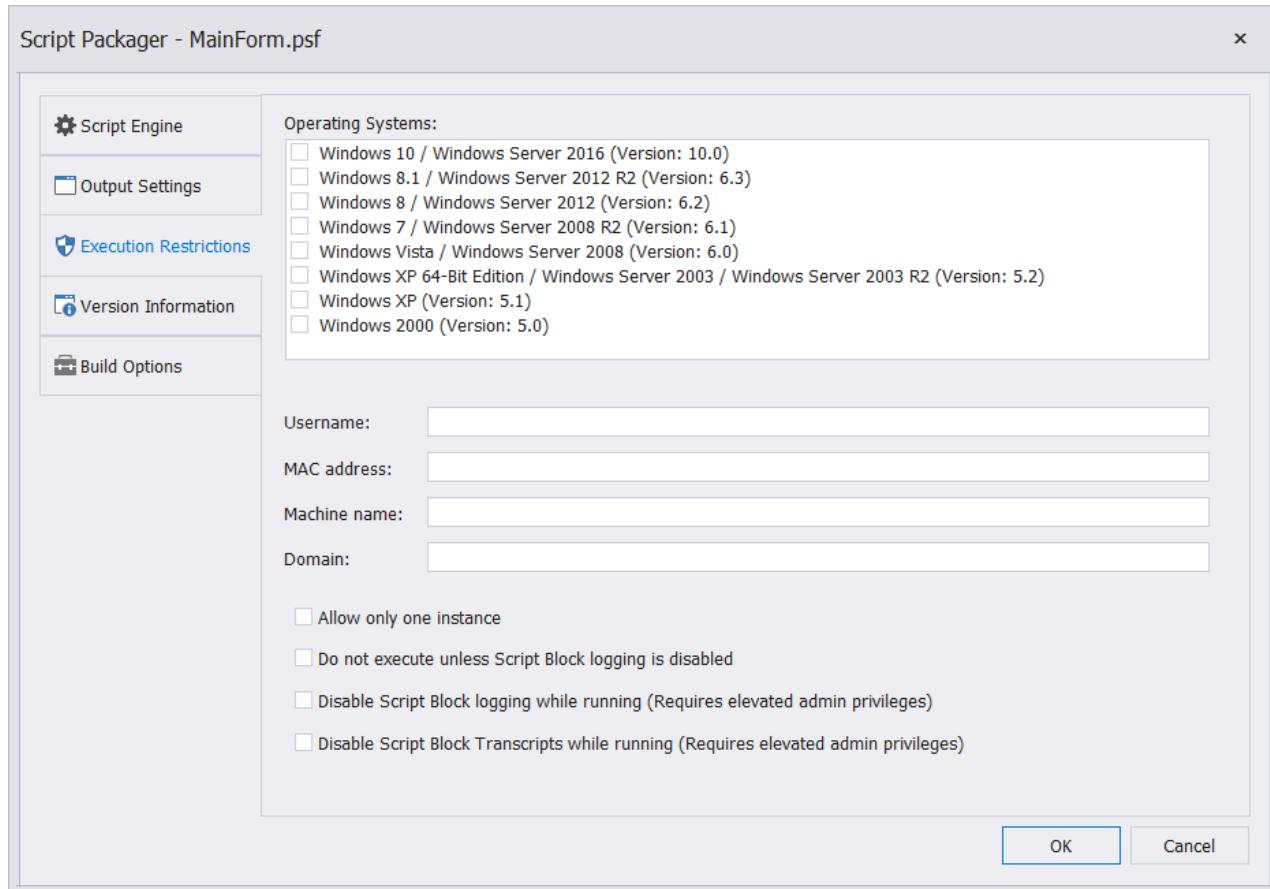
Using this process, Starter.exe will launch and use the specified Admin credentials, and then your script will run with elevation.

 Depending on your local settings, you may get a prompt to allow your script to modify your system, but it will not prompt you for actual credentials.

## Execution Restrictions

---

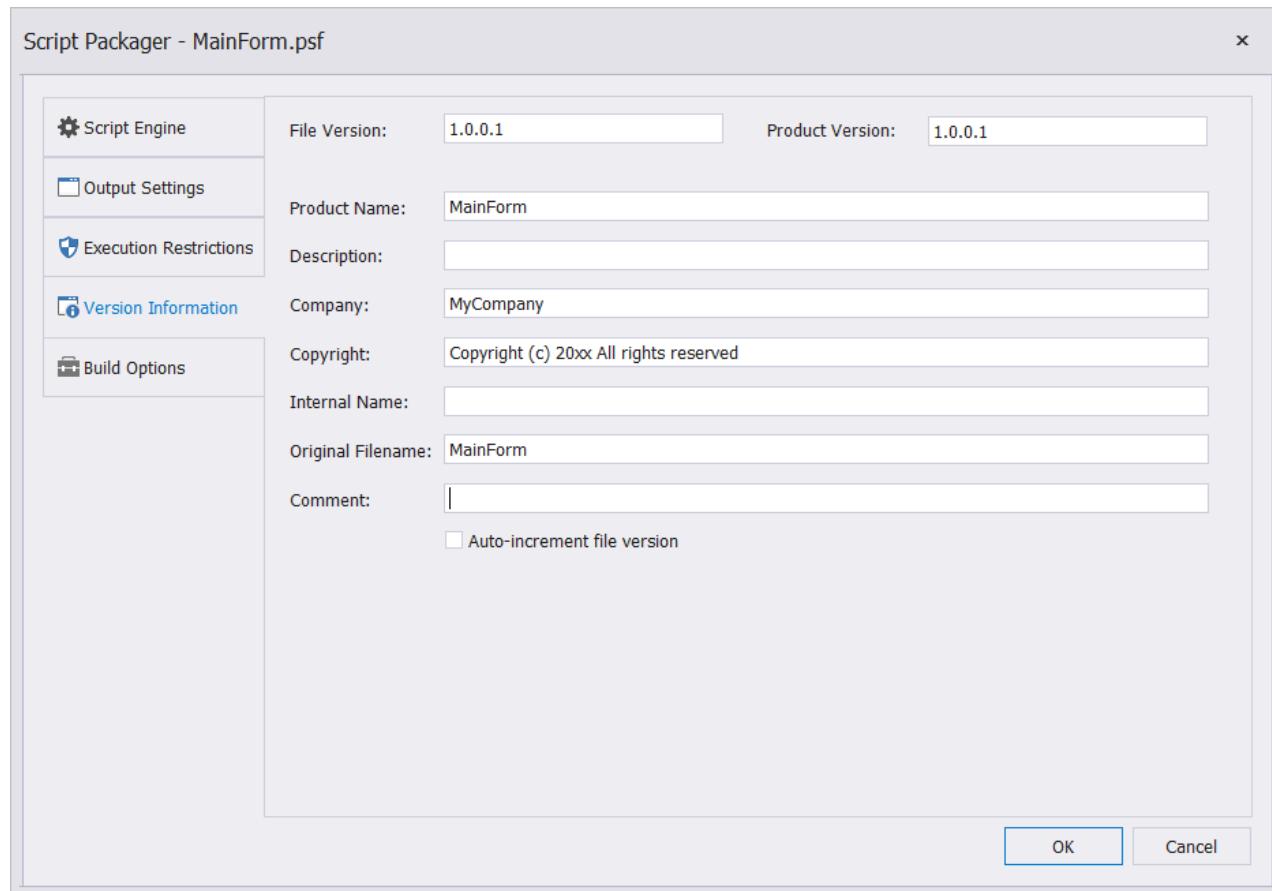
Use the Execution Restrictions to limit the environment in which the package runs.



- i** When restricted to a specific version, the executables display the expected and encountered versions in the error message.

## Version Information

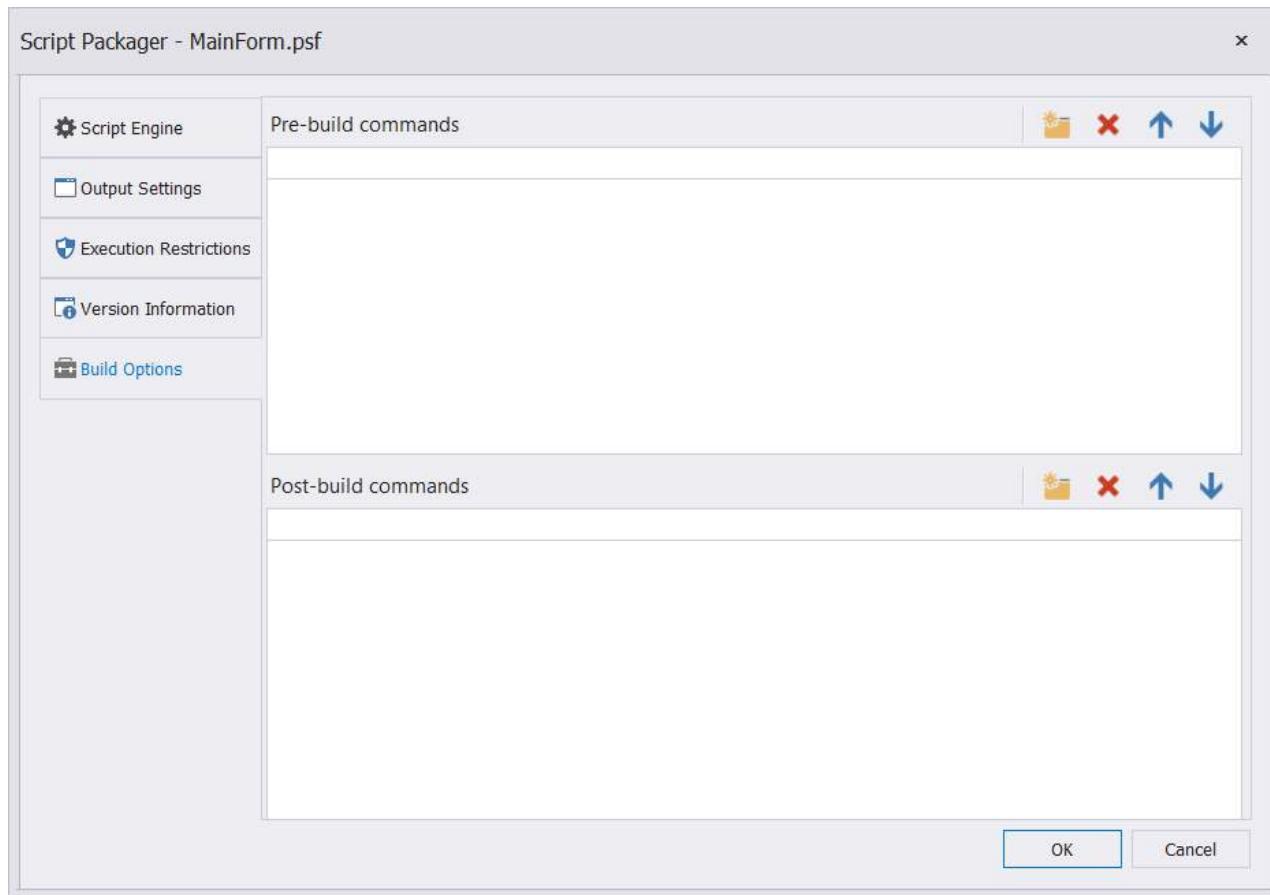
Use the Version Information settings to specify characteristics of the current version of the executable file.



! The version number must be in #.#.#.# format.

## Build Options

Use the Build Options to define custom commands to run before or after packaging.



Use the four buttons at the top-right of each section to manage the pre- and post-packaging commands:



From left to right:

-  **Add File** - Browses for a file / exe.
-  **Remove** - Removes the command.
-  **Move Up** - Moves the command up in the order.
-  **Move Down** - Moves the command down in the order.

 The commands will be executed in the sequence defined, and one after the other, rather than in parallel.

## 11 Source Control Integration

PowerShell Studio provides a number of source control options, including a Universal Version Control system that integrates with command-line tools such as [Git](#)<sup>[302]</sup>, or integrating with a Microsoft Source Code Control Integration ([MS SCCI](#)<sup>[305]</sup>) software provider.

### 11.1 Universal Version Control

The Universal Version Control system allows configuration of any source control provider with command-line tools. The current support scope includes the Git source control system. Support will be expanded to include other providers.

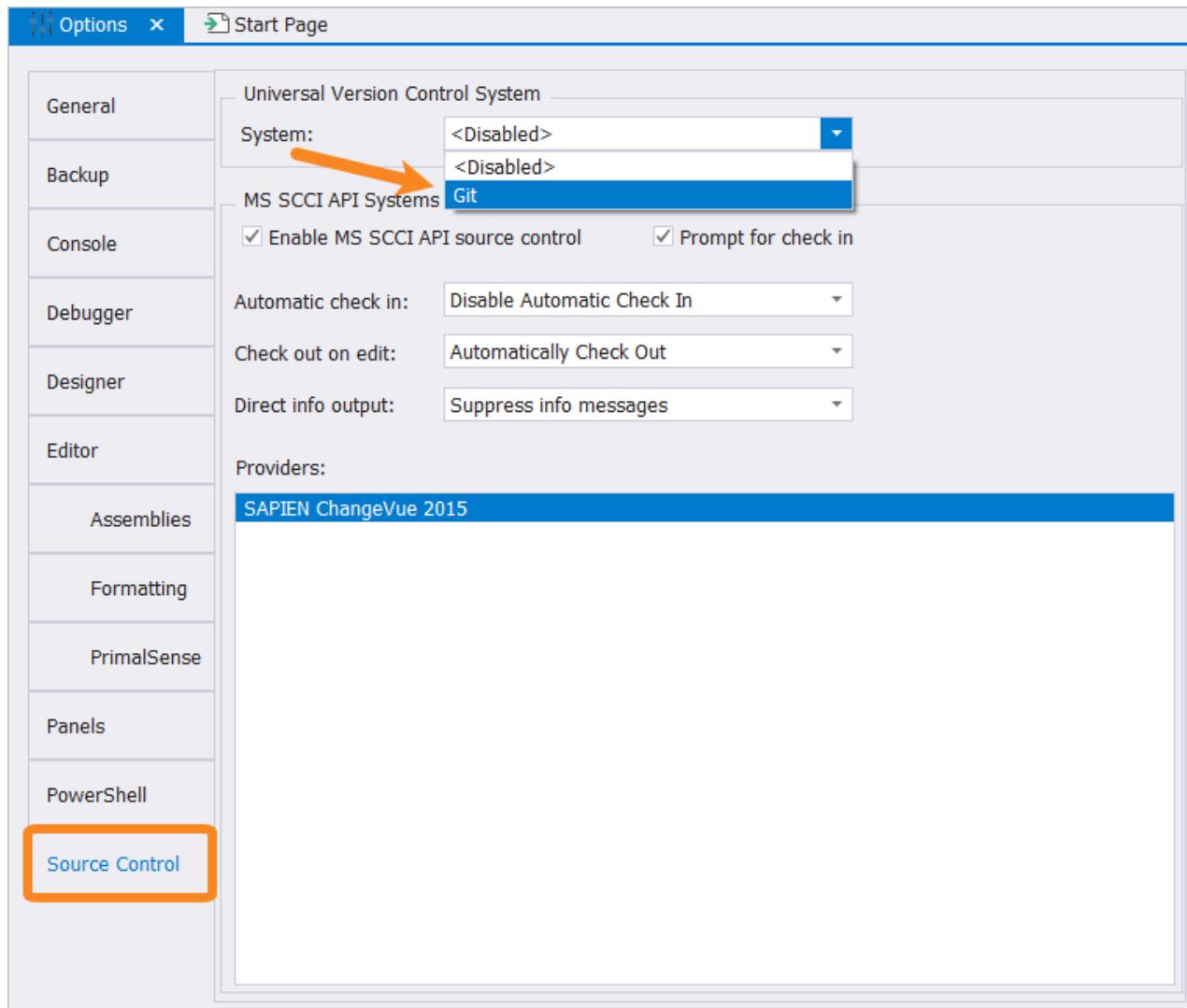
#### Using Git

---

##### To enable Git support

---

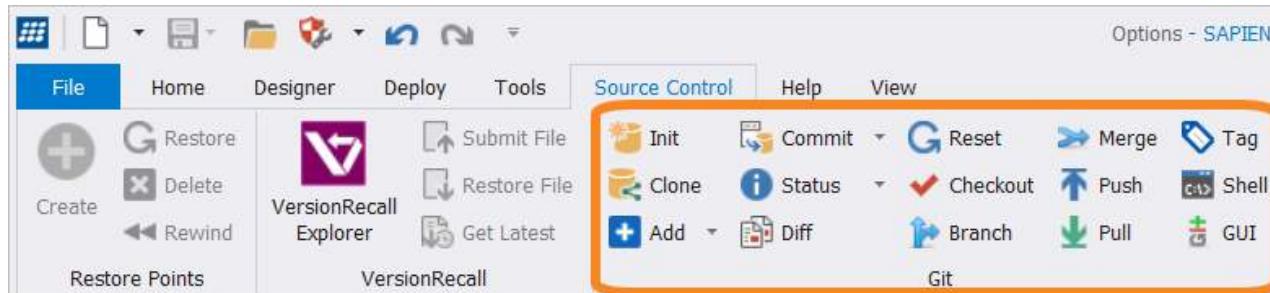
Go to **File > Options > Source Control**. In the System drop-down list, select **Git**:



- i** To disable the Universal Version Control feature, select <Disabled> from the System drop-down.

## Git Commands

Once enabled, the preconfigured Git commands will appear in the **Source Control** tab on the ribbon:



- **Init**

Initialize a Git repository in the current folder.

- **Clone**

Create a clone of a remote repository.

- **Add**

Add a file to a repository.

- **Add All**

Add a file or all files in the folder to a repository.

- **Commit**

Commit a change to a repository.

- **Commit All**

Commit all changes to a repository.

- **Status**

Get the status of the current file.

- **Status All**

Get the status of the current file or all files in the folder.

- **Diff**

Show the difference for the current file.

- **Reset**

Rewinds history (files + commits) back to the previous commits.

- **Checkout**

Switch branches or restore working tree files.

- **Branch**

Create a new branch.

- **Merge**

Merge the specified branch.

- **Push**

Upload the local repository content to a remote repository.

- **Pull**

Fetch and download content from a local repository.

- **Tag**

Create a tag for the current repository.

- **Shell**

Launch a Git command shell.

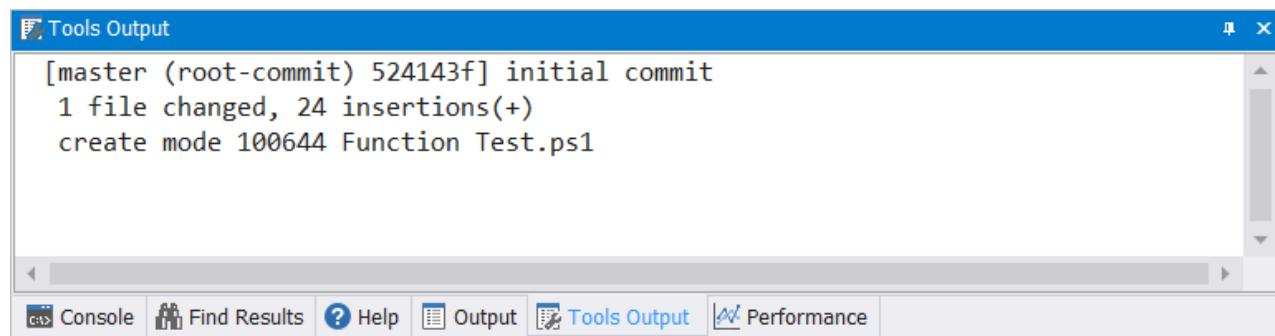
- **GUI**

Launch the Git GUI tool.

You will be prompted if a value is required to execute the command. For example, when you select the Git **Commit** command, a commit message is required:



- Info Output from Git will be displayed in the Tools Output panel:



## 11.2 Microsoft Source Code Control Integration

Your source control software must either be [VersionRecall from SAPIEN Technologies](#), or your source control provider must provide an SSAPI-compatible client, such as Microsoft Visual Source Safe.

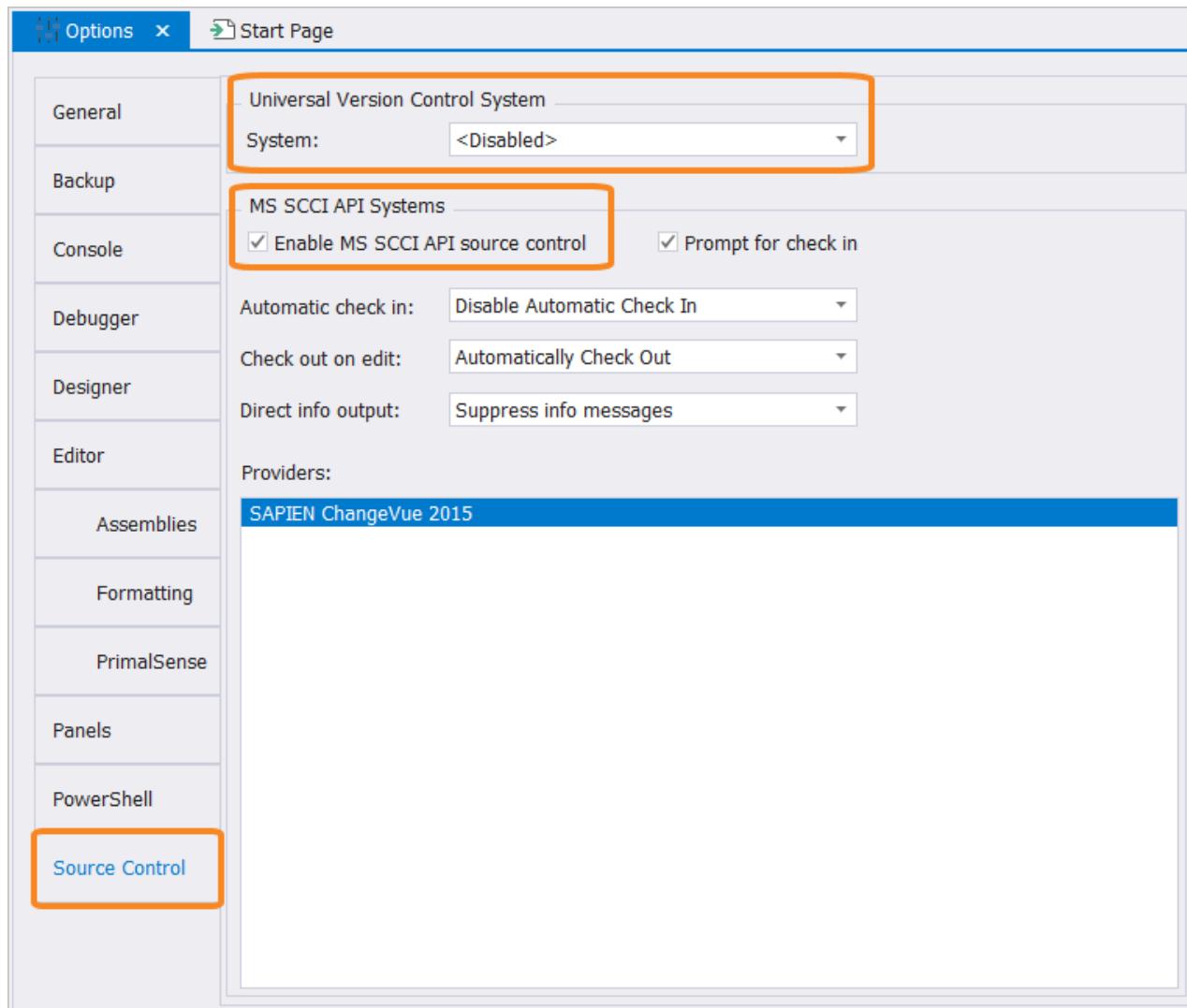
### Configuring Source Control Integration

- Info Before configuring PowerShell Studio for source control, you must install your source control software's client.

#### To configure source control integration

Go to File > Options > Source Control:

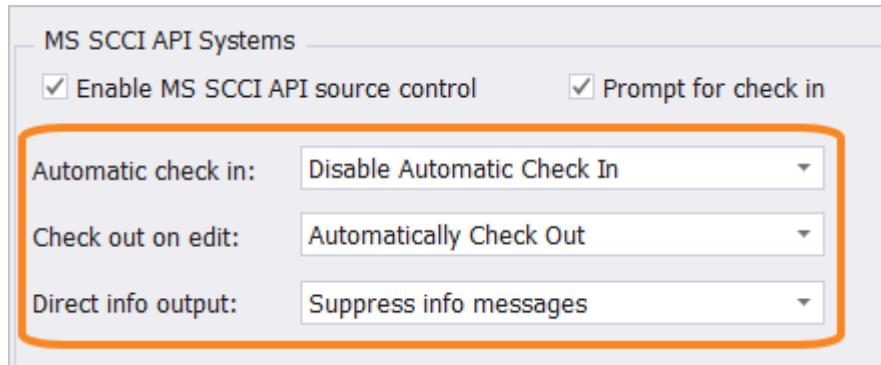
1. Make sure that the Universal Version Control feature is <Disabled>.
2. Select Enable MS SCCI API source control.



PowerShell Studio will automatically detect the presence of the source control client and automatically display it in the **Providers** list box. If your source control client does not appear, then shutdown and restart PowerShell Studio.

- ! Your source control provider must be displayed in the Provider list; if it is not, then source control is not properly installed and will not be available to PowerShell Studio.

After enabling source control you can configure the options however you like, including prompting before checking files out, automatic check-in, and directing output messages:



Source control settings are configured in [Options and Settings > Source Control](#) [374].

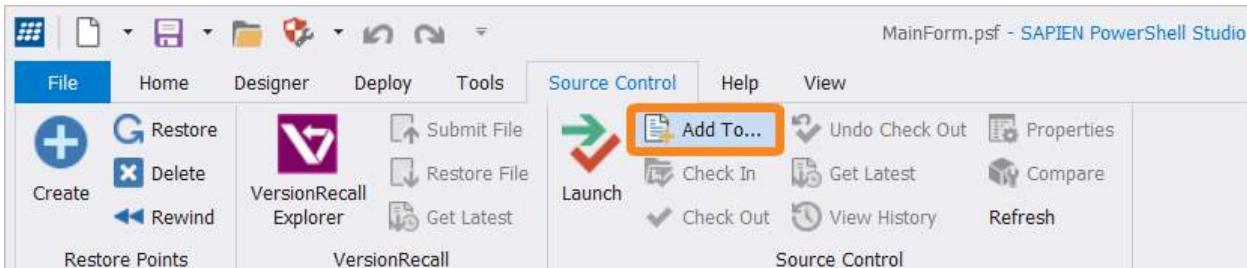
## Using Source Control

- i** PowerShell Studio does not provide source control capability—it simply integrates with the features of your compatible source control software. Some features described here may not be available in your software, or may work somewhat differently.

Before a file can be managed through source control, it must first be added.

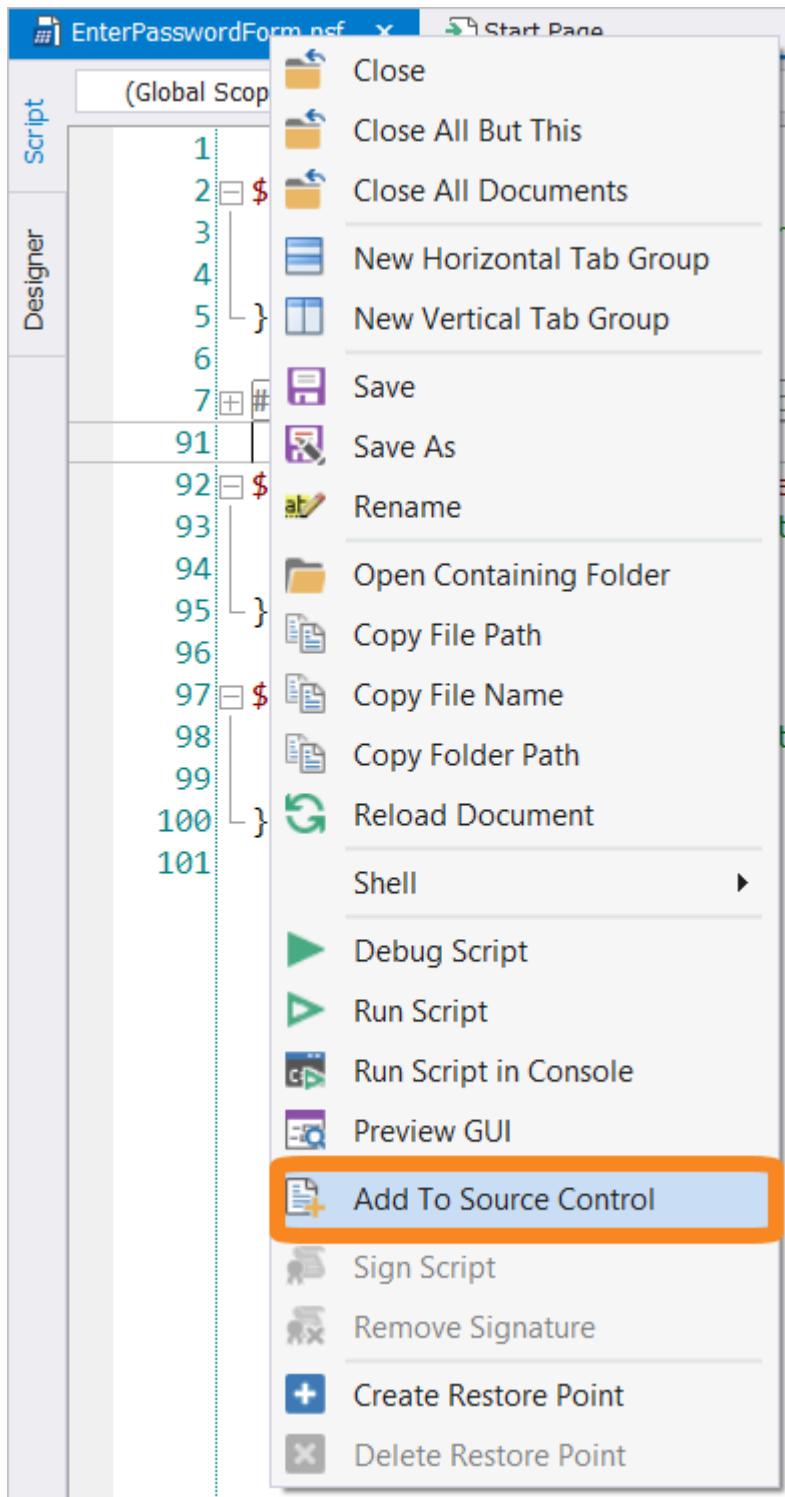
### To add a file to source control

- With the file open, on the **Source Control** tab > select **Add To...**:



-OR-

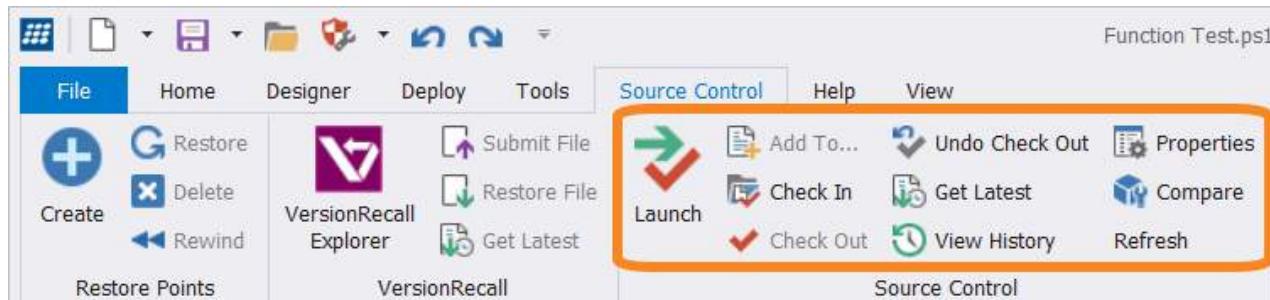
- Right-click the file name tab and select **Add To Source Control**:



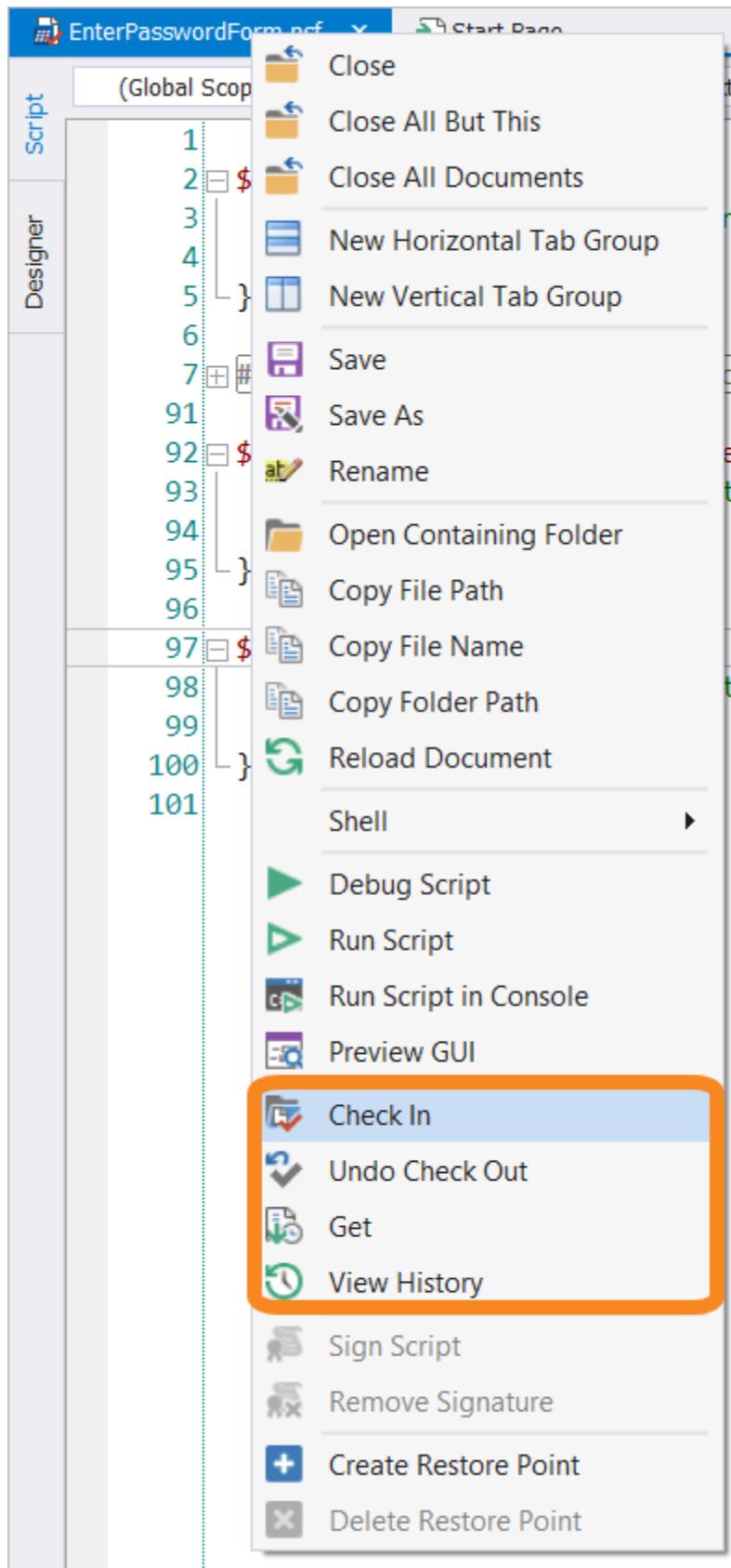
! You must first save unsaved scripts before they can be added. If you do not, PowerShell Studio will prompt you to save the file first.

- i** Your source control software governs the add process and may prompt you for login credentials, a location for the script, or other information.

Once added, scripts can be checked in or out using the buttons on the Source Control ribbon:



Source control functions are also available on the file context menu:



## Source Control Commands

---

Some of these source control options may not be available, or may work differently, depending on your source control provider:

- **Launch**  
Launches the source control software.
- **Add To...**  
Add the current document to a source control database.
- **Check In**  
Checks in the changes of the current document into the source control database.
- **Check Out**  
Checks out the current document for editing from the source control database.
- **Undo Check Out**  
Restores the file to the last checked in version.
- **Get Latest**  
Get the latest version of the document from the source control database.
- **View History**  
View the past versions of the current document.
- **Properties**  
View the source control properties of the current document.
- **Compare**  
Compares the active document to a previous version.
- **Refresh**  
Refresh the source control status.

## 12 ScriptMerge

ScriptMerge is a stand-alone application shipped with PowerShell Studio that compares files and folders and applies differences to either of the two compared items.

### 12.1 Running ScriptMerge

ScriptMerge can be started from the Windows Start Menu, or from within the PrimalScript and PowerShell Studio applications.

#### To start ScriptMerge from the Windows Start Menu

- In the Windows Start Menu, select SAPIEN Technologies, Inc. > ScriptMerge:

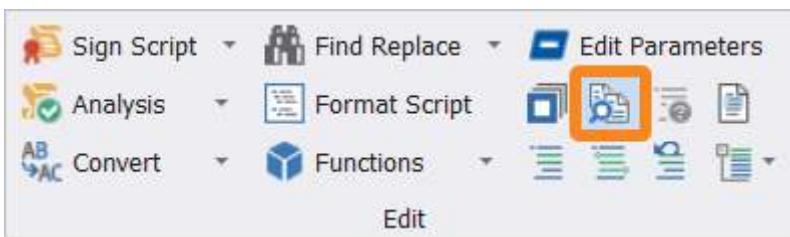


#### To start ScriptMerge from PrimalScript

1. In PrimalScript click the View tab > then in the Panels section, check the Tools box.
2. In the Tools Browser click SAPIEN Tools > then click the ScriptMerge icon.

#### To start ScriptMerge from PowerShell Studio

- In PowerShell Studio, open two files to compare > then click Home > in the Edit section, click the Compare Files button:



## 12.2 Comparing Files

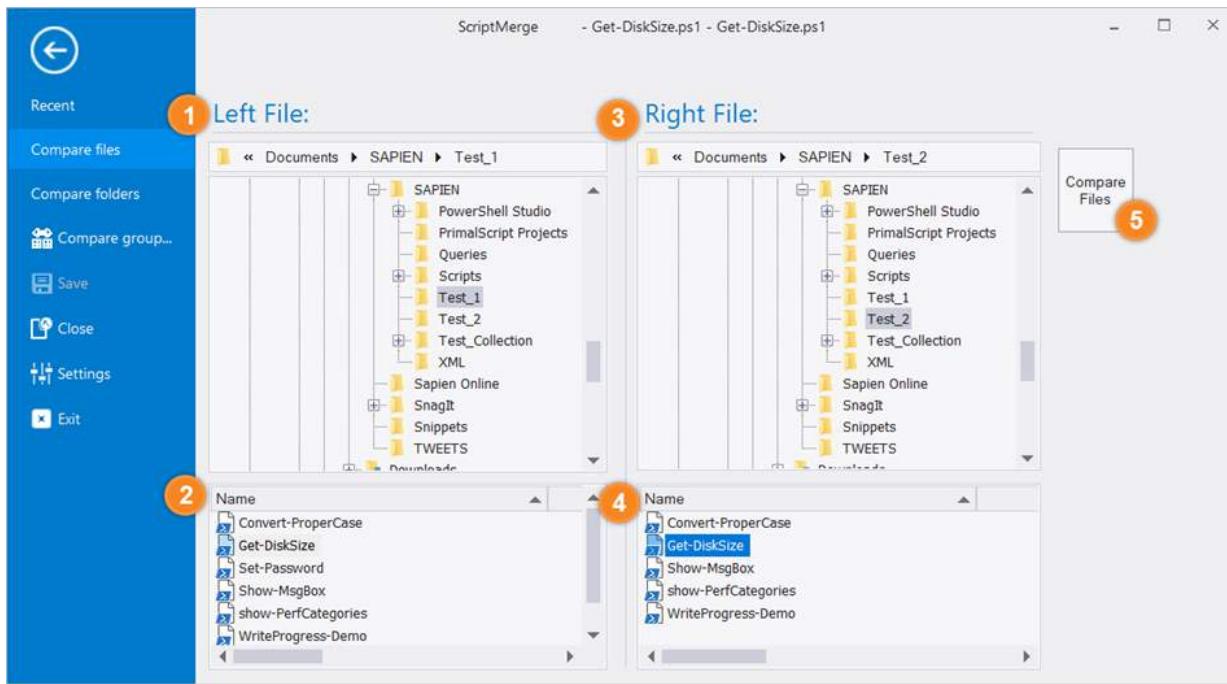
ScriptMerge compares files side-by-side and highlights the differences.

#### To compare files

Click File > Compare files:

1. In the Left File window, navigate to the folder.
2. Select a file in the Name window below.
3. In the Right File window, navigate to the folder.
4. Select the other file to compare in the Name window below.

5. Click Compare Files.



When the files are first opened, ScriptMerge displays the differences as gray, light yellow, and dark yellow colored lines. The current difference—in this case the first difference—is highlighted in varying shades of red:

- Light yellow indicates words that have changed.
- Dark yellow indicates a line that contains a change.
- Dark grey lines indicate a line that was deleted.

```

1
2 Param ($Computer = "localhost")
3 $cldisks = get-wmiobject Win32_LogicalDisk -comput
4 " Device ID   Type      Size(M)    Free
5 ForEach ($disk in $cldisks)
6 {
7   $drivetype=$disk.drivetype
8   Switch ($drivetype)
9   {
10     2 ($drivetype="FDD")
11     3 ($drivetype="HDD")
12     5 ($drivetype="CD ")
13   }
14
15 " (0)          (1)          (2,15:n)  (3,15:n)" -f $:
16 }

# Get Disk Info
Param ($Computer = "localhost")
$cldisks = get-wmiobject Win32_LogicalDisk -comput
" Device ID   Type      Size(mb)    Free
ForEach ($Disk in $cldisks)
{
  $Drivetype=$Disk.Drivetype
  Switch ($Drivetype)
  {
    2 ($Drivetype="FDD")
    3 ($Drivetype="HDD" )
    4 ($Drivetype="Net" )
    5 ($Drivetype="CD " )
  }
}

```

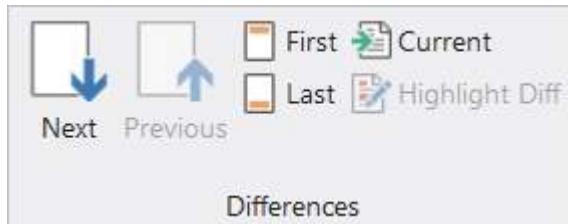
Line: 1-2 Col: -1/-1 Ch: -1/-1      Line: 2 Col: 1/16 Ch: 1/16

To see the highlighting in action, change a line and save it. The changes will be reflected in the comparison.

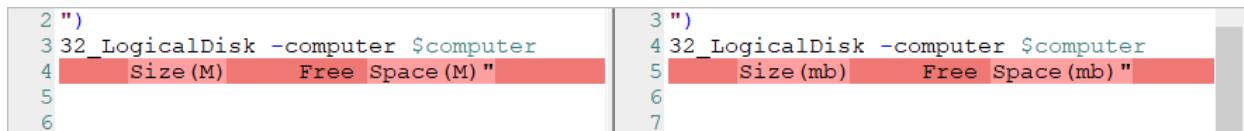
 You can customize the comparison differences coloring in File > Settings > Merge Options > Color Options.

### To step through the differences

- In the Differences section, click **Next** and **Previous** to go back and forth through the differences:

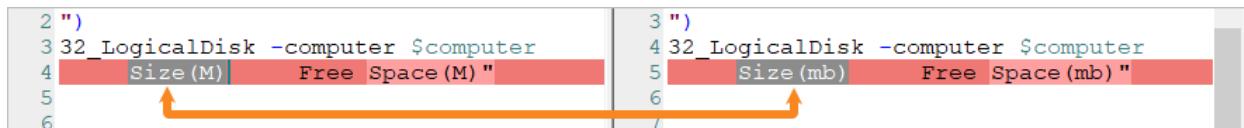


The current difference is highlighted in different shades of red:

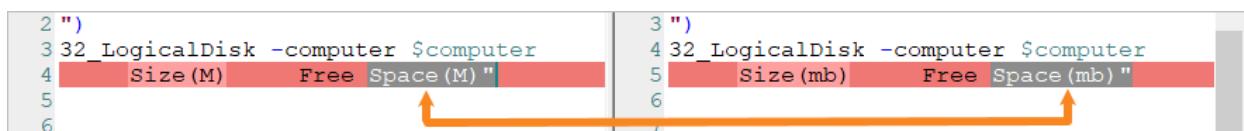


```
2 ")
3 32_LogicalDisk -computer $computer
4     Size(M)     Free Space(M) "
5
6
7
```

Click **Highlight Diff** to add extra emphasis to the changed elements for the current difference:



```
2 ")
3 32_LogicalDisk -computer $computer
4     Size(M)     Free Space(M) "
5
6
```



```
2 ")
3 32_LogicalDisk -computer $computer
4     Size(mb)     Free Space(mb) "
5
6
```

### To merge the differences

- In the Merge section, select **Right** or **Left**:



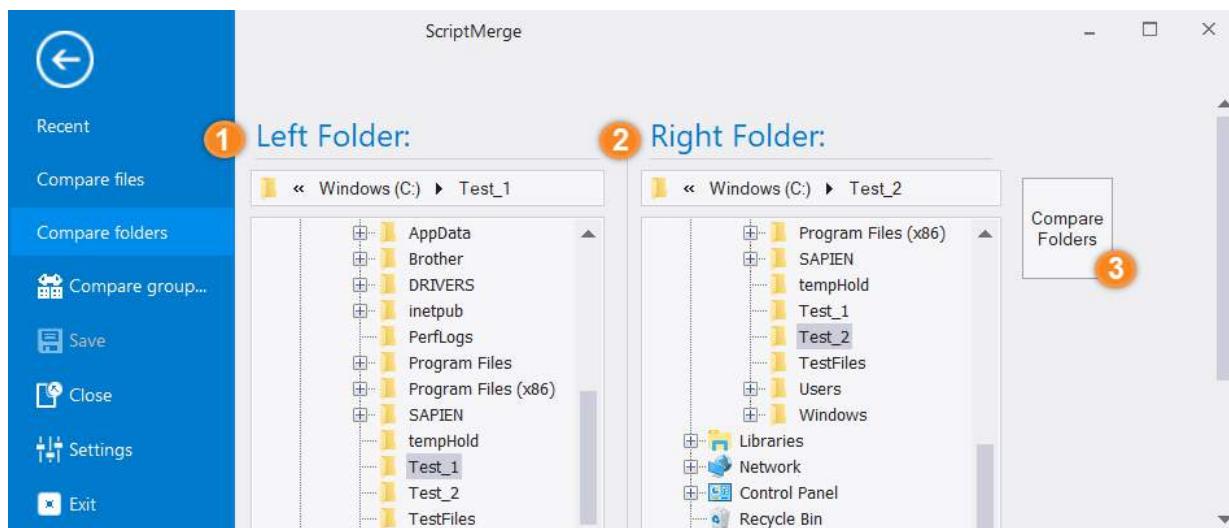
## 12.3 Comparing Folders

ScriptMerge compares folders side-by-side and highlights the differences.

### To compare folders

Click File > Compare folders:

1. In the Left Folder window, select a folder.
2. In the Right Folder window, select a folder.
3. Click Compare Folders.



ScriptMerge compares the files in each folder and their contents. The results show which folders have files in both locations, or if the folders have files in only one location. If the folders have files in both locations, ScriptMerge indicates if the files are different or identical.

Files that exist in both locations but are different are marked with a red whole page icon:

Filename	Comparison result	Left Date	Right Date	Extension
...				
Convert-ProperCase.ps1	Files are different	6/14/2018 11:35:09 AM	* 6/14/2018 11:36:00 AM	ps1
Get-DiskSize.ps1	Files are different	6/14/2018 1:29:04 PM	* 6/14/2018 1:32:35 PM	ps1
Show_MsgBox.ps1	Identical	6/14/2018 11:43:03 AM	6/14/2018 11:43:03 AM	ps1
show-PerfCategories.ps1	Identical	6/14/2018 11:41:58 AM	6/14/2018 11:41:58 AM	ps1
WriteProgress_Demo.ps1	Identical	6/14/2018 11:39:45 AM	6/14/2018 11:39:45 AM	ps1

**Red whole page icon =  
Files are different**

Files that are identical in both locations are marked with a blue whole page icon:

Filename	Comparison result	Left Date	Right Date	Extension
...				
Convert-ProperCase.ps1	Files are different	6/14/2018 11:35:09 AM	* 6/14/2018 11:36:00 AM	.ps1
Get-DiskSize.ps1	Files are different	6/14/2018 1:29:04 PM	* 6/14/2018 1:32:35 PM	.ps1
Show-MsgBox.ps1	Identical	6/14/2018 11:43:03 AM	6/14/2018 11:43:03 AM	.ps1
Stop-PerfCategories.ps1	Identical	6/14/2018 11:41:58 AM	6/14/2018 11:41:58 AM	.ps1
WriteProgress-Demo.ps1	Identical	6/14/2018 11:39:45 AM	6/14/2018 11:39:45 AM	.ps1

**Blue whole page icon =  
Files are identical**

Files that are only in one folder are marked with a blue half-page icon. The icons reflect the folder location: left half-page icons are in the Left Folder; right half-page icons are in the Right Folder:

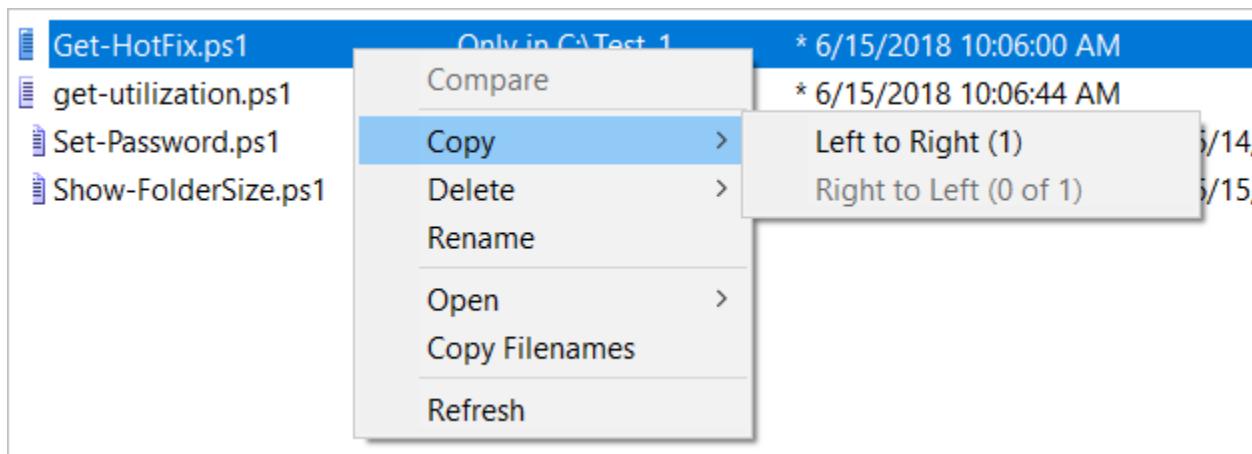
Left: C:\Test\_1\
Right: C:\Test\_2\

Filename	Comparison result	Left Date	Right Date	Extension
...				
Get-HotFix.ps1	Only in C:\Test_1	* 6/15/2018 10:06:00 AM		.ps1
get-utilization.ps1	Only in C:\Test_1	* 6/15/2018 10:06:44 AM		.ps1
Set-Password.ps1	Only in C:\Test_2		* 6/14/2018 11:38:29 AM	.ps1
Show-FolderSize.ps1	Only in C:\Test_2		* 6/15/2018 10:06:21 AM	.ps1

**Blue half-page icons reflect the folder location:  
- Left half-page icon = file is in the Left Folder.  
- Right half-page icon = file is in the Right Folder.**

### To replace a file in one folder with the file in the other folder

- In the Merge section, select Right or Left.
- OR-
- Right-click the file and select Copy (Left to Right or Right to Left):



## 12.4 Comparing Groups

You can group pairs of files and then easily open the group to compare. This feature is useful for repeated comparison of the same files.

### To create and open a group

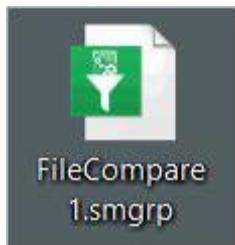
---

1. Create a text file with the file pairs listed as follows:

File1|File2 separated by the pipe symbol ( | ).

Example: C:\Users\Me\Documents\SAPIEN\script1.ps1|C:\Users\Me\Documents\SAPIEN\script2.ps1

2. Save the file as <*filename*>.smgrp (smgrp = ScriptMerge Group):

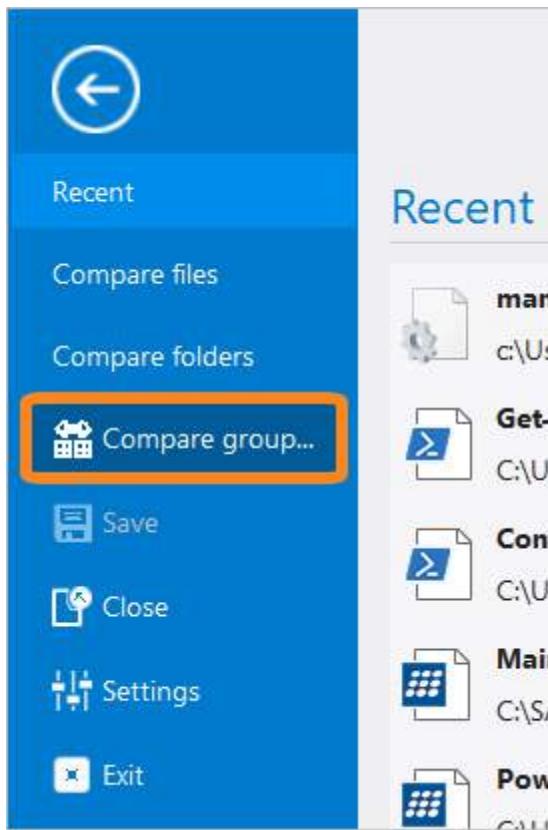


3. Open the group file and ScriptMerge will open the contained pairs at the position of the first difference:

- Double-click the group file.

-OR-

- In ScriptMerge select **File > Compare group**, then navigate to the group file location:



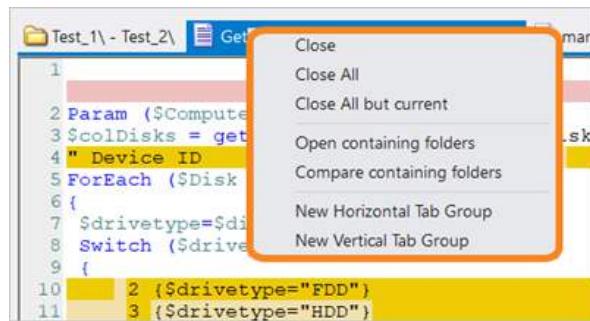
## 12.5 Context Menu Options

The ScriptMerge context menu options will vary depending on if you are comparing files or folders.

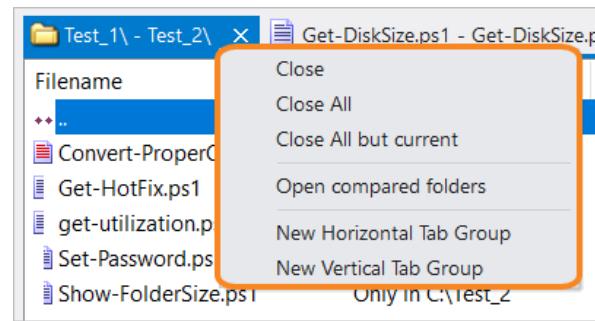
### To access the context menu options

- Right-click on the file comparison or folder comparison tab:

Compare Files - Context Menu



Compare Folders - Context Menu



- Close

Closes the highlighted tab.

- **Close All**

Closes all of the tabs.

- **Close All but current**

Closes all tabs except for the highlighted tab.

- **Open containing folders (file comparison only)**

Opens two Windows Explorer instances with the compared files selected.

- **Compare containing folders (file comparison only)**

Compares the files in each underlying folder, and also the file contents.

- **Open compared folders (folder comparison only)**

Opens two Windows Explorer instances, one for each compared folder.

- **New Horizontal Tab Group**

Moves the selected tab to a separate horizontal tab group.

- **New Vertical Tab Group**

Moves the selected tab to a separate vertical tab group.

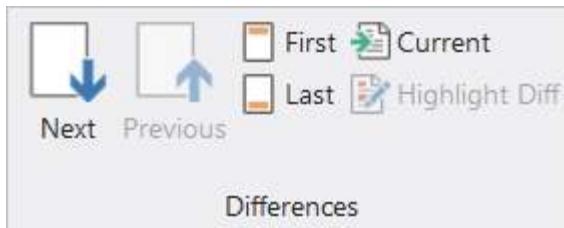
- **Move to Previous Tab Group**

Moves the selected tab back to the original tab group.

👉 Tab groups are especially useful when you have a folder comparison open and also a number of files compared. Move the folder comparison to its own tabbed group so that it remains visible while you compare the files in the folders.

## 12.6 Navigating Between Differences

The **Differences** section of the ribbon provides buttons to help you move between differences in a file or folder:



- **Next (Ctrl+Down)**

Moves forward to the next difference.

- **Previous (Ctrl+Up)**

Moves back to the previous difference.

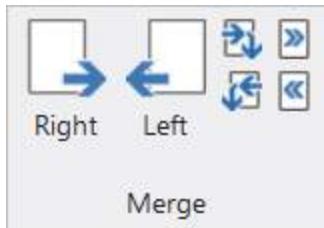
- **First (Ctrl+H)**

Moves to the first difference.

- **Last (Ctrl+E)**  
Moves to the last difference.
- **Current (Ctrl+Enter)**  
Scrolls the code window to the current difference.
- **Highlight Diff**  
Adds extra emphasis to the changed elements in code files.

## 12.7 Reconciling Differences

The **Merge** section of the ribbon provides buttons to help you copy from one file or folder to another:



- **Right (Ctrl+Right)**  
Copies the current selection from the left to the right file or folder.
- **Left (Ctrl+Left)**  
Copies the current selection from the right to the left file or folder.
- **Right, Next Diff**  
Copies the current selection from the left to the right and advances to the next difference.
- **Left, Next Diff**  
Copies the current selection from the right to the left and advances to the next difference.
- **All Right**  
Copies all differences from the left to the right file or folder.
- **All Left**  
Copies all differences from the right to the left file or folder.

## 12.8 Signing Scripts

ScriptMerge allows you to handle digital signatures in your files.

You can re-sign scripts or remove signatures from the ribbon buttons:



- **Sign Left**

Sign the script file displayed on the left.

- **Sign Right**

Sign the script file displayed on the right.

- **Sign Both**

Sign both script files.

- **Remove Left**

Remove the signature from the script file displayed on the left.

- **Remove Right**

Remove the signature from the script file displayed on the right.

- **Remove Both**

Remove the signatures from both script files.

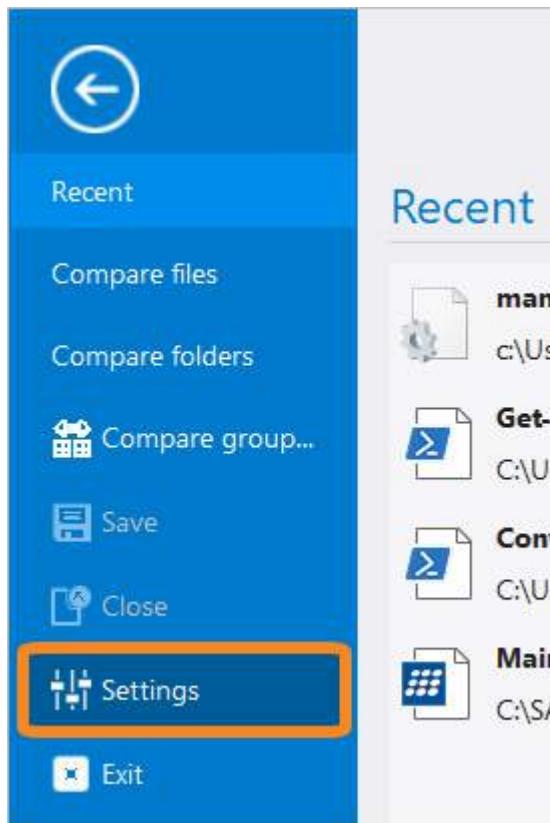
## 12.9 ScriptMerge Settings

You can adjust some ScriptMerge tool settings, such as keyboard shortcuts and Quick Access Toolbar buttons. You can also change the highlight colors for comparisons, and toggle some compare options.

---

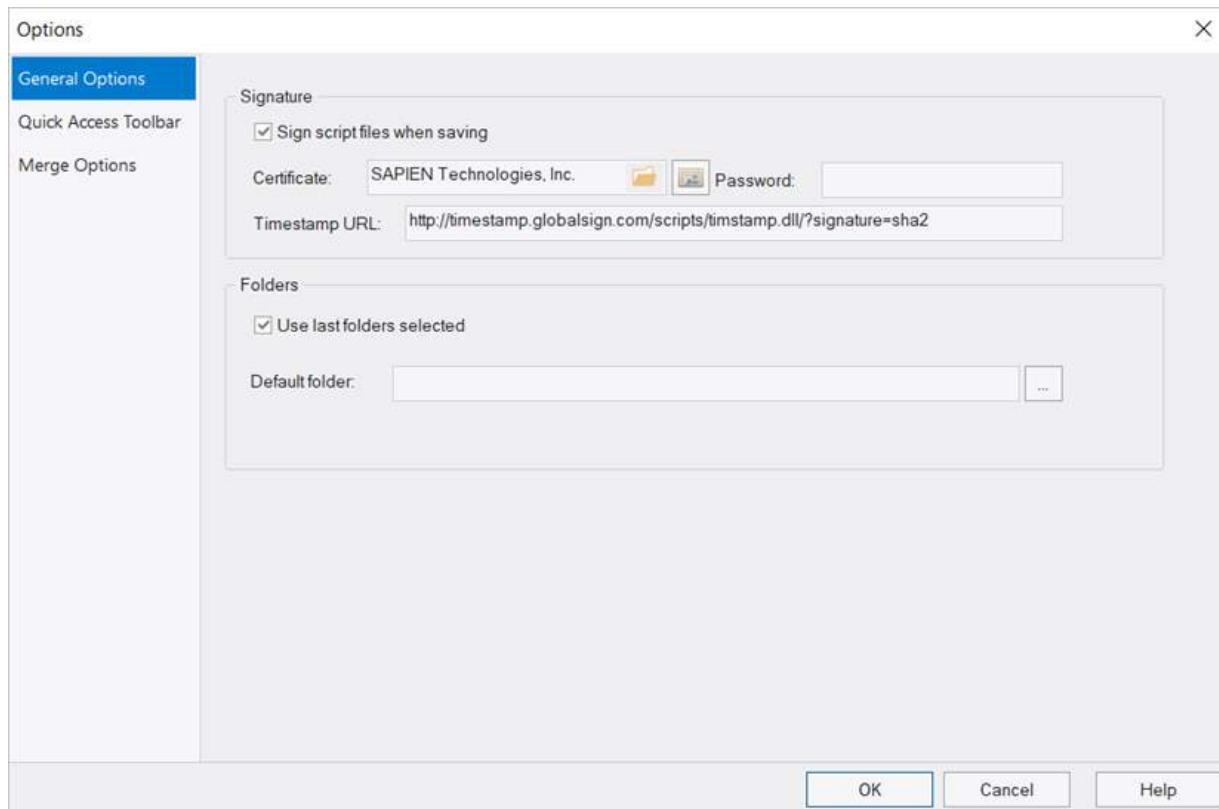
### To access the ScriptMerge options

- Select **File > Settings:**



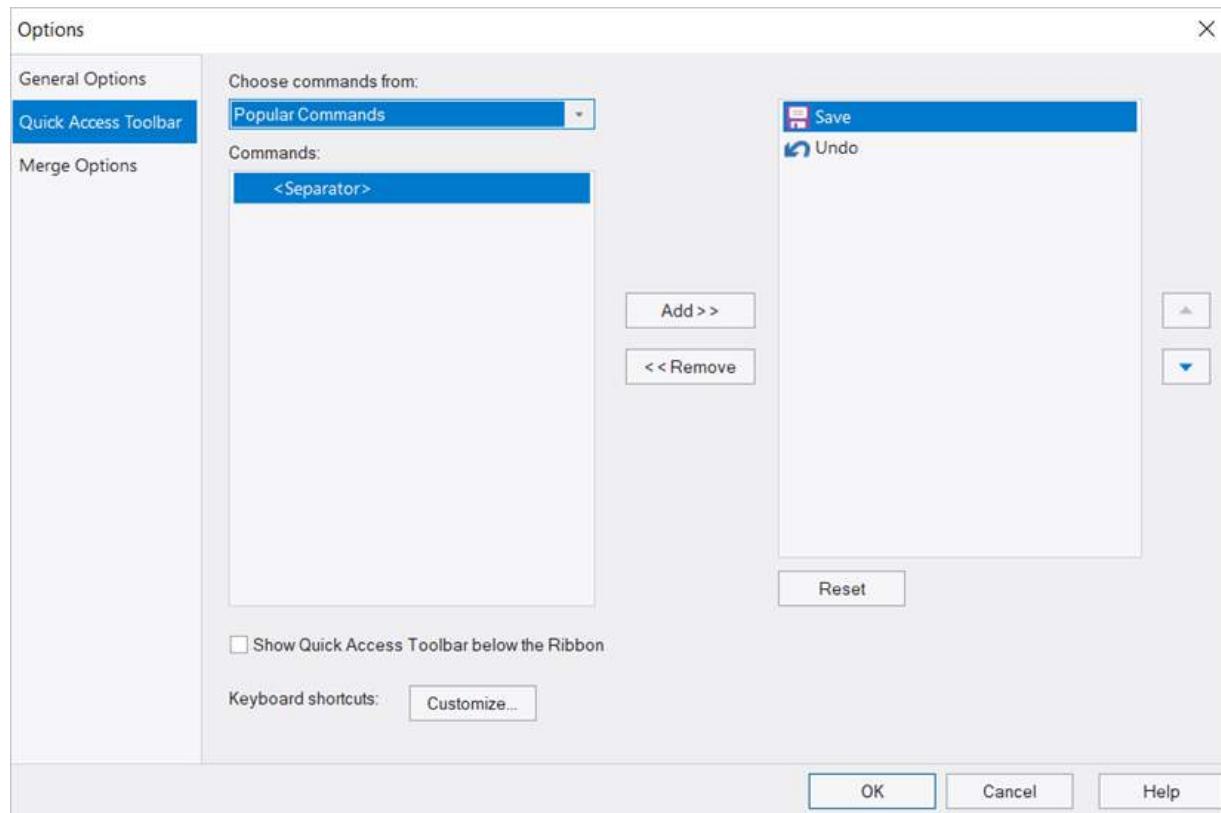
The following settings can be adjusted:

- **General Options**
  - Sign script files when saving.
  - Use last folders selected.



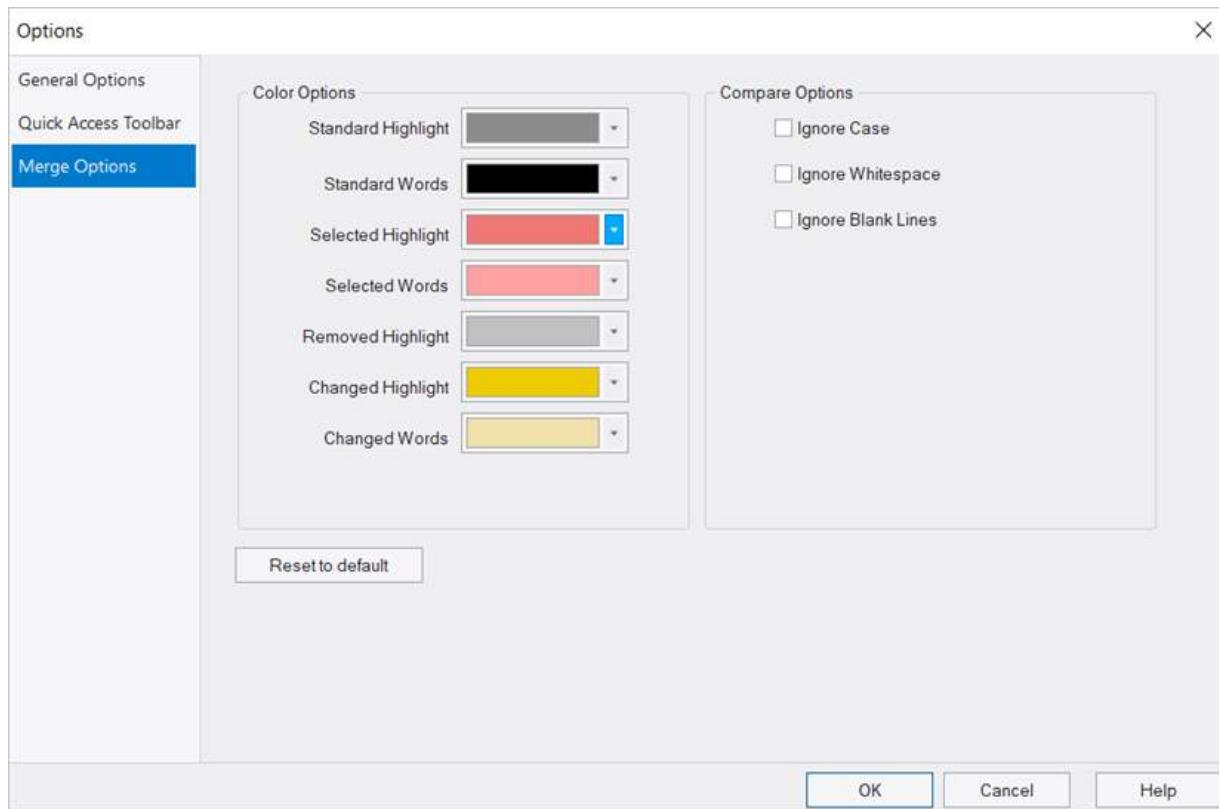
- **Quick Access Toolbar**

- Add, Remove, Reset toolbar buttons.
- Show Quick Access Toolbar below the Ribbon.
- Customize Keyboard Shortcuts.



- **Merge Options**

- Select comparison color options.
- Toggle compare options for Case, Whitespace, and Blank Lines.



## 13 Snippet Editor

PowerShell Studio provides a collection of snippets to help you complete common coding tasks quickly. You can use the [Snippet Editor](#)<sup>326</sup> to easily edit and create snippets.

### About Snippets

---

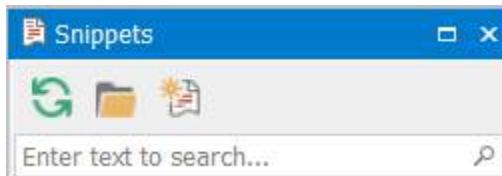
Snippets are small pieces of reusable code that can be quickly inserted into your scripts, thus saving you time and reducing errors. This piece, or "snippet" of code, can vary from a full-fledged function to a simple single line statement. Snippets come in a variety of languages such VBScript, PowerShell, C#, etc.

PrimalScript and PowerShell Studio come with extensive libraries of reusable code snippets. You can also save any text or code block as a snippet to automate code development. Snippets can include placeholders; PrimalScript and PowerShell Studio will prompt you to supply values for these when you use the snippet.

### Snippets Panel

---

Use the Snippets panel to access and manage snippets:



#### To access the Snippets panel:

---

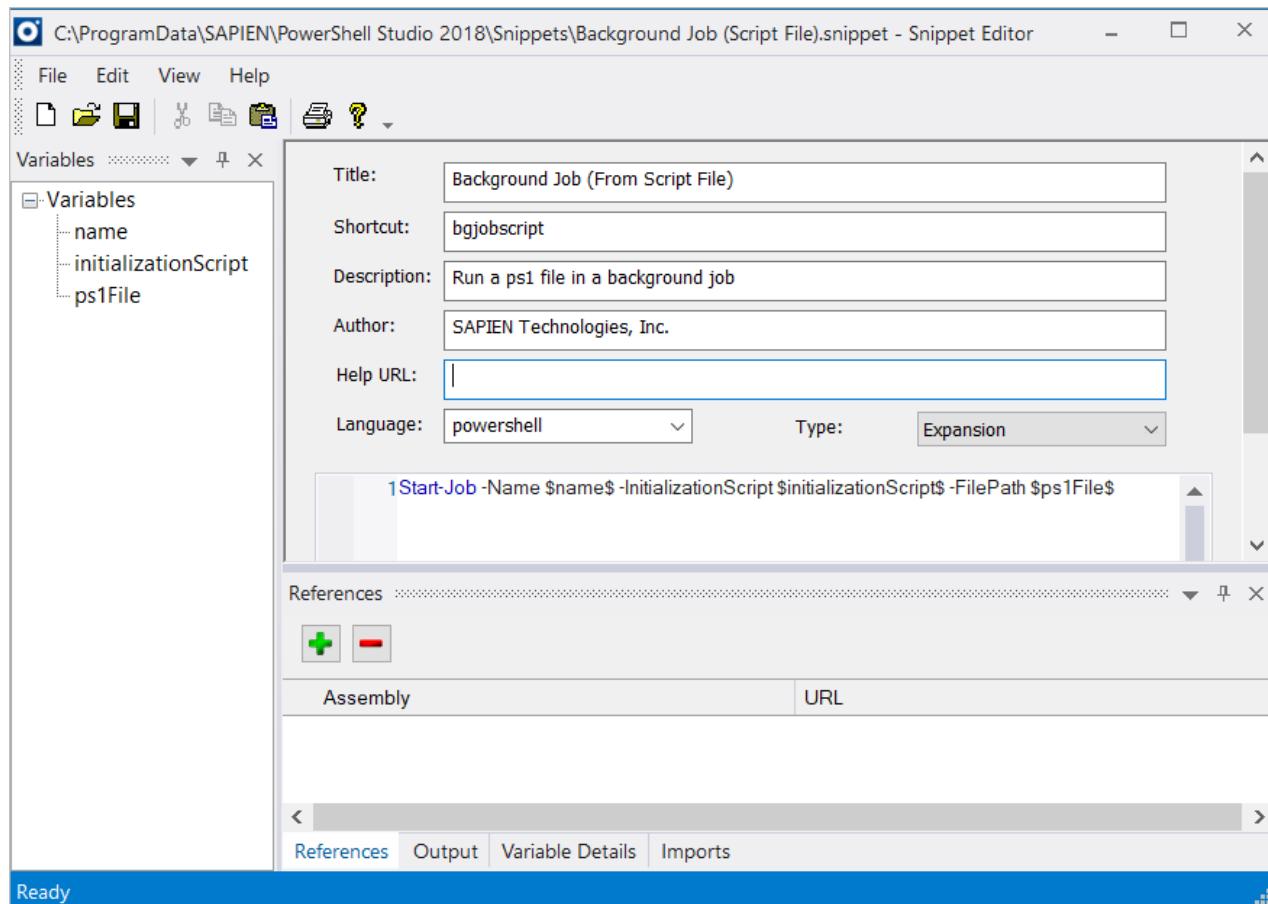
- On the Home ribbon, in the Windows section, select Snippets from the Panels drop-down menu.
- OR-
- Chorded keyboard shortcut: Press **Ctrl+Alt+P**, release, then press **S**

### Snippet Editor

---

The Snippet Editor is a self-contained program within PrimalScript and PowerShell Studio that supports multiple programming languages. Using the Snippet Editor is a fast and easy way to edit existing snippets and to create your own.

The Snippet Editor will launch when you edit an existing snippet or create a new snippet:



*The Snippet Editor*

## Snippet Properties

The top section of the Snippet Editor allows you to enter the following snippet properties:

- **Title**

The name of the snippet.

- **Shortcut**

The text you need to type in the code editor to invoke the snippet.

- **Description**

A short description of the snippet explaining what it does.

- **Author**

The snippet author details.

- **Help URL**

A link to help information. This will be displayed in the code editor.

- **Language**

Set to 'powershell' for snippets used in PowerShell Studio.

Set to the appropriate language for snippets used in PrimalScript.

- **Type**

This setting defines how the snippet will be displayed in the code editor (inserted into the code, or surrounding existing code). The options are:

- Expansion**

Select this if your snippet is intended to be simply inserted into code.

- Surrounds With**

Select this if your snippet can surround existing code.

- Both**

Select this if your snippet can be used both ways.

The selection you choose for the **Type** property will dictate the menu options available when you insert the snippet in the code editor:



- **Insert Snippet...**

Only displays snippets where the 'Type' property is defined as **Expansion** or **Both**.

- **Surround With Snippet...**

Only displays snippets where the 'Type' property is defined as **Surrounds With** or **Both**.

**!** If you choose a 'Type' value of **Surrounds With** or **Both** you must include the `$selected$` placeholder variable\* somewhere in your snippet code body, otherwise you may overwrite user code when your snippet is used:

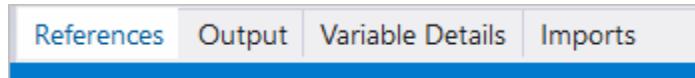
(\* Refer to the [Built-in Placeholder Variables](#) section below for more information about the placeholder variables provided with the Snippet Editor.)

A screenshot of the Snippet Editor's configuration dialog. The snippet is titled 'DoWhile'. It has a 'Shortcut' of 'DoWhile', a 'Description' of 'Creates a Do While loop', and an 'Author' of 'SAPIEN Technologies, Inc.'. The 'Language' is set to 'PowerShell' and the 'Type' is 'Both'. In the code body, line 3 contains the placeholder variable '\$selected\$'. The code body is as follows:

```
1
2do {
3 $selected$$end$
4} while ($conditions)
5
```

## Snippet Windows

The tabs at the bottom of the Snippet Editor provide more configuration options for your snippets:



- **References**

This section tells PrimalScript or PowerShell Studio what dependencies your snippet has. The assemblies you list here will be loaded into PrimalScript or PowerShell Studio when you use the snippet.

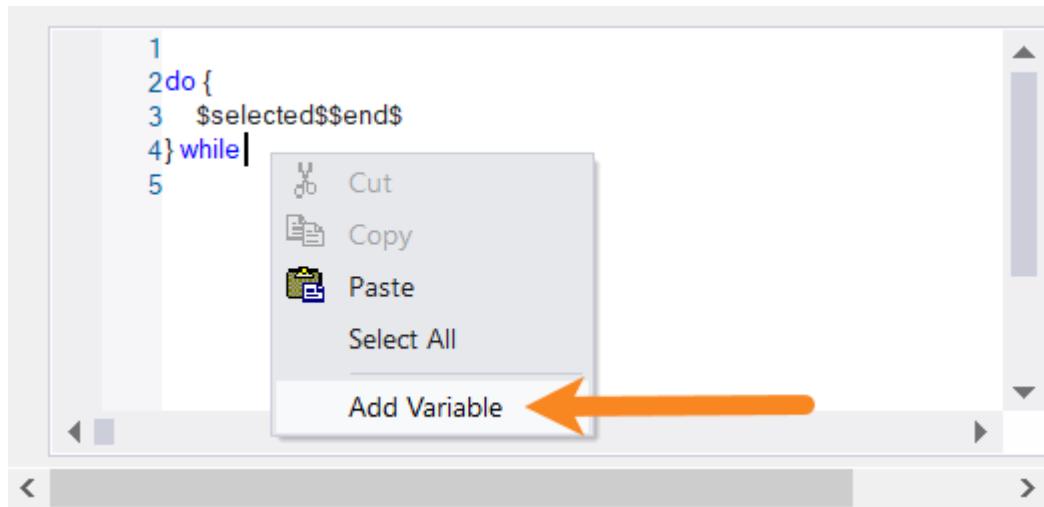
- **Output**

This section is not used for PowerShell snippets.

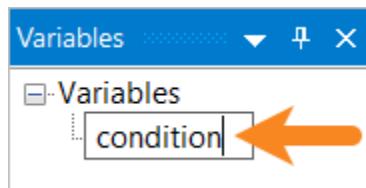
- **Variable Details**

Before you configure the variable details you must add a placeholder variable to a snippet:

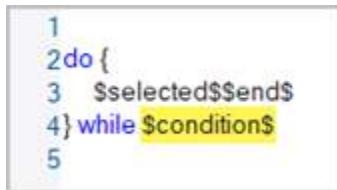
1. Position your cursor in the snippet code editor where you want to insert a variable, then right-click and select **Add Variable**:



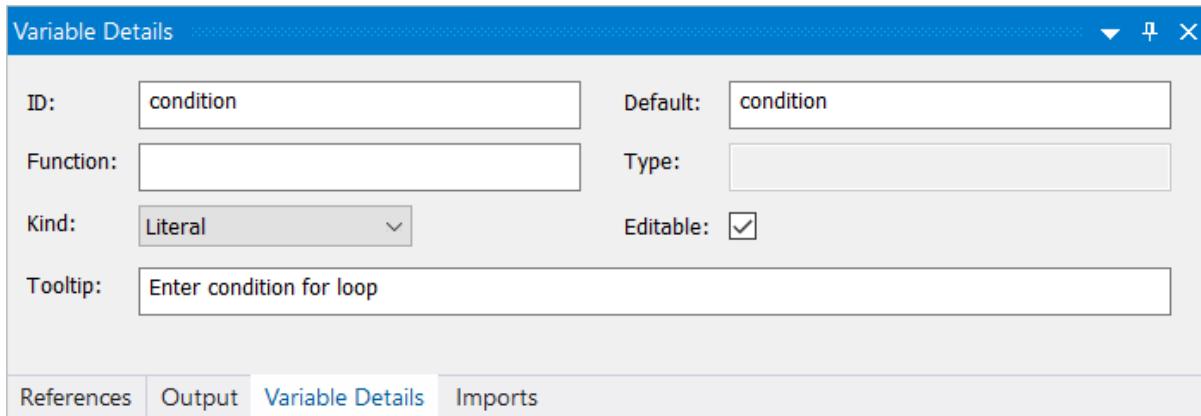
2. Name the variable:



3. The variable will appear in the snippet as \$<variable>\$ (e.g., \$condition\$):



4. Configure additional variable properties:



- **ID**

The variable name.

- **Default**

A default value if required.

- **Function**

Not used in PowerShell snippets.

- **Type**

Not used in PowerShell snippets.

- **Kind**

Not used in PowerShell snippets.

- **Editable**

Not used in PowerShell snippets.

- **ToolTip**

Provides some text explaining the purpose of the variable. This helps the snippet user understand how to complete the snippet. PrimalScript or PowerShell Studio will display these tool-tips as the user navigates between the placeholders in the code editor.

- **Imports**

This section is not used for PowerShell snippets.

## Built-in Placeholder Variables

The Snippet Editor provides two built-in placeholder variables:

- **\$selected\$**

Allows you to merge code from the code editor into your snippet when it is used. For example, you could create a snippet called ExtractFunction containing this code:

```
1function $function-name$()  
2{  
3    $selected$  
4}
```

Now you can highlight lines of code and use this snippet to refactor them into a reusable function.

- **\$end\$**

Specifies where the cursor should be placed when a snippet is inserted into the code editor.

## 14 Options and Settings

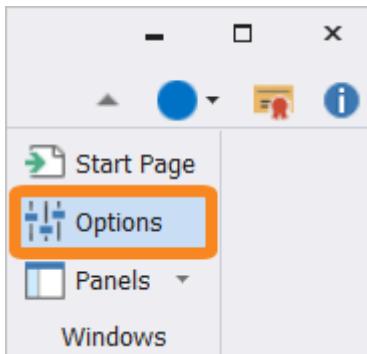
The Options dialog contains all of the main configuration settings for PowerShell Studio. This section provides an overview of the available settings.

### 14.1 Accessing the Options

There are a number of ways to access the PowerShell Studio Options from the ribbon.

#### To access the PowerShell Studio configuration options

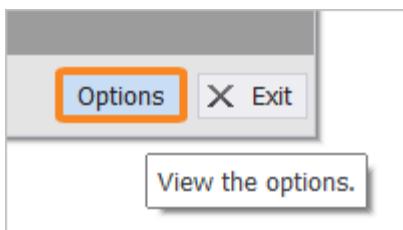
- On the **Home** tab > in the Windows section (far right), select **Options**:



**i** The Windows section of the ribbon might be compressed on smaller screens, such as tablets.

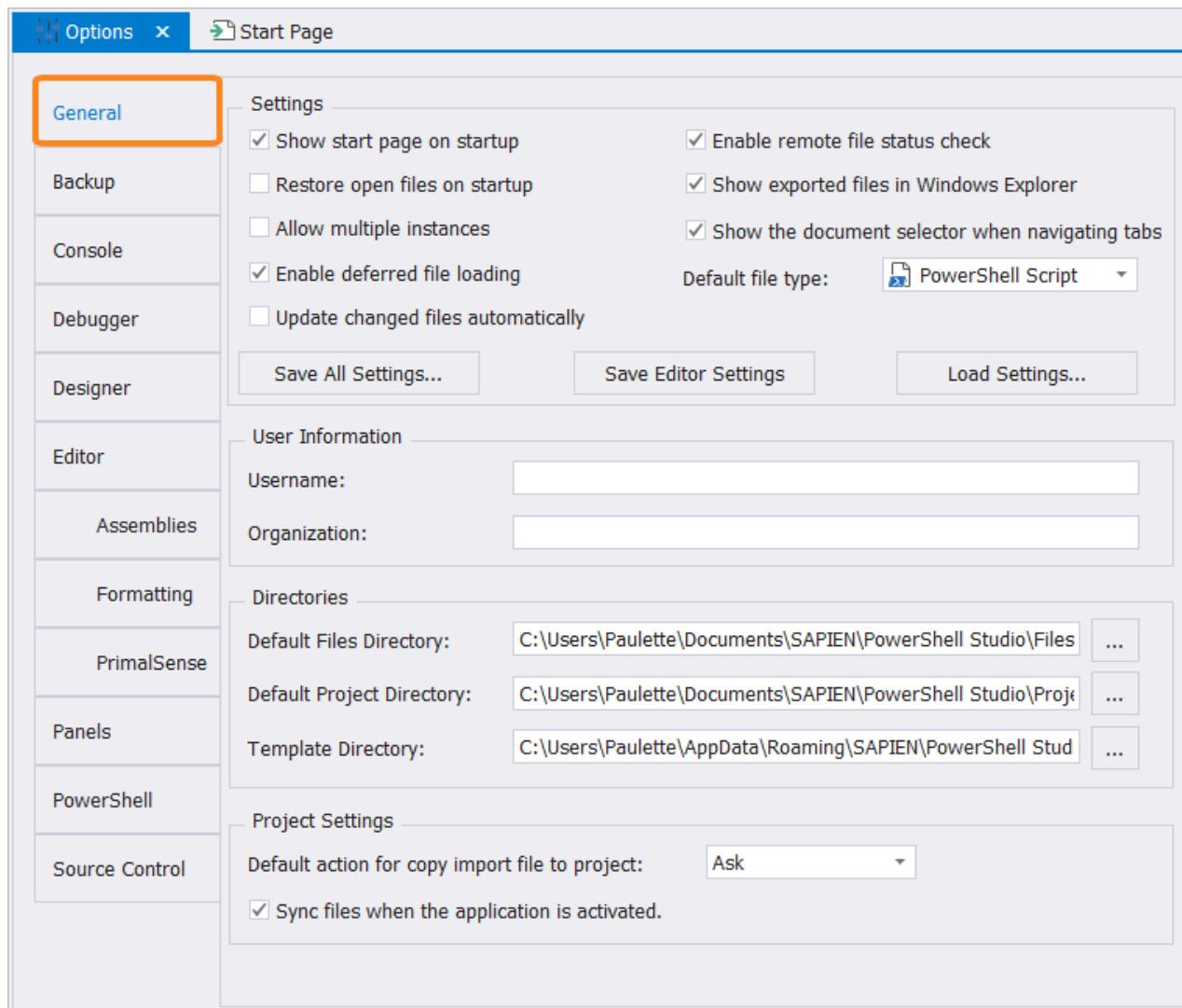
-OR-

- Select the **File** tab > on the bottom-right, select the **Options** button:



### 14.2 General

This topic covers the settings available in Options > General.



## In this section

- [Settings](#) [333]
  - [Saving Settings](#) [334]
- [User Information](#) [335]
- [Directories](#) [335]
- [Project Settings](#)

## Settings

- **Show start page on startup**

Enables or disables the [Start page](#) [19].

- **Restore open files on startup**

Reopens the files that were open when PowerShell Studio was last shut down.

- **Allow multiple instances**

Allows for more than one copy of PowerShell Studio running at once.

- **Enable deferred file loading**

Loads files on demand. Deferred loading improves overall performance when loading large groups of files while reducing memory consumption and load times.

- **Update changed files automatically**

Automatically reloads files that are modified externally. PowerShell Studio will still prompt if the file has any unsaved changes.

- **Enable remote file status check**

PowerShell Studio will warn you if a file has been edited outside of its editor. Checking remote files can cause PowerShell Studio to slow down. This option allows you disable remote file checking if needed.

- **Show exported files in Windows Explorer**

After exporting files from PowerShell Studio, launch Windows Explorer focused on the export folder.

- **Show the document selector when navigating tabs**

Controls the behavior of navigating between documents (*Ctrl+Tab*).

- **Enabled**

Cycle between documents using the Document Selector. The documents are displayed in activation order instead of tab order.

- **Disabled**

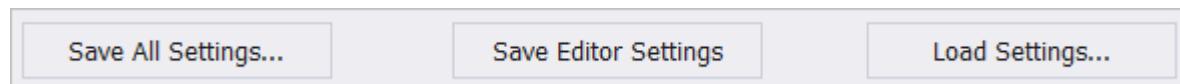
Cycle through the document tabs without opening the Document Selector.

- **Default file type**

Choose the default file type when a new document is created. Choose from PowerShell Script or Form.

## Saving Settings

There are three buttons at the bottom of the Settings section that control the saving and loading of PowerShell Studio settings:



These settings can be used to back-up or load your settings, and they can also be shared with others for standardization purposes.

- **Save All Settings...**

This button saves all application settings, and it is recommended that you regularly export the settings. By default, the settings are saved to:

*Documents\SAPIEN\PowerShell Studio\Files\PowerShellStudio.Settings.xml*

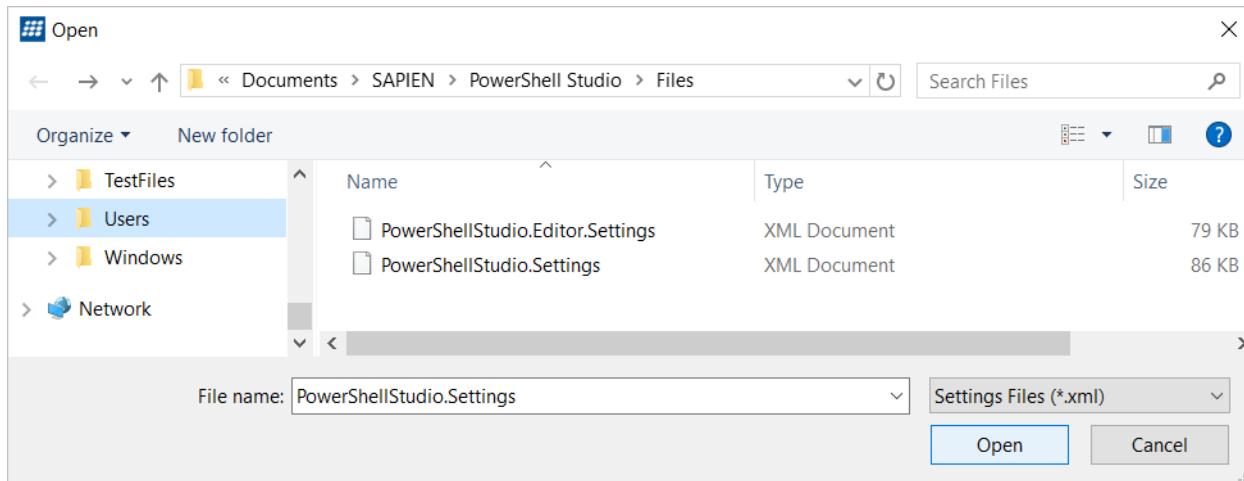
- **Save Editor Settings**

This button saves only the editor settings. By default, the settings are saved to:

*Documents\SAPIEN\PowerShell Studio\Files\PowerShellStudio.Editor.Settings.xml*

- **Load Settings...**

This button loads settings that were previously saved.



i You will need to restart PowerShell Studio for the changes to take effect.

## User Information

- **Username**

Enter the user name. This information will be used in your templates.

- **Organization**

Enter the organization name. This information will be used in your templates.

## Directories

- **Default Files Directory**

Specifies the folders where scripts are saved.

- **Default Project Directory**

Specifies the folders where projects are saved.

- **Template Directory**

Specifies the folders where user templates are saved.

## Project Settings

- **Default action for copy import file to project**

When importing an existing file into a project, PowerShell Studio can make a copy of the file and add it to the project folder or create a link to the original file.

- **Ask**

Ask to copy import file.

- **Copy**

Automatically copy import file.

- **Never**

Never copy import file.

- **Sync files when the application is activated**

Allows you to trigger project file synchronization when the application regains focus (activated).

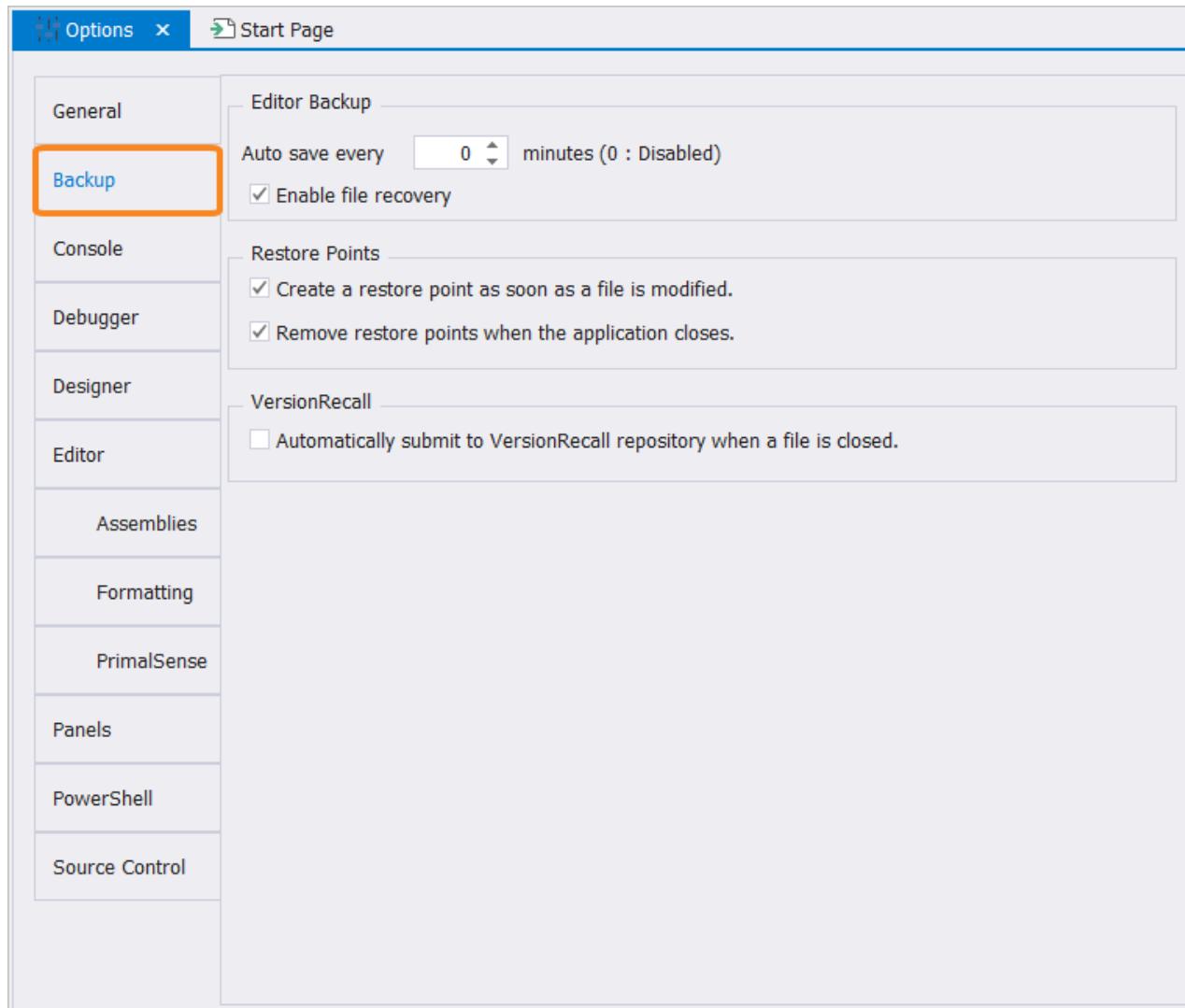
You also need to have a project open with file synchronization enabled (*Synchronized = True*):



*Project sync on activate* ensures that the application can detect changes when you are making modifications to the project's folder structure outside of PowerShell Studio.

## 14.3 Backup

This topic covers the settings available in Options > Backup.



## In this section

- [Editor Backup](#) 337
- [Restore Points](#) 338
- [VersionRecall](#)

## Editor Backup

- **Auto save every [ ] minutes**

The frequency in minutes that files will be auto-saved.

- **Enable file recovery**

Activates the File Recovery system to recover modified files when the application is restarted after an unexpected closure.

 Changes are not committed for recovered files until the user explicitly saves the modified document.

## Restore Points

---

- **Create a restore point as soon as a file is modified**

Create a restore point when you modify a file to allow you to easily undo all changes. The automatic restore point is only created once, at the time of first edit (start of a session). When you create a permanent restore point, it only stores one. This is meant as a quick recovery tool and not as a versioning tool. So if you work on your script for a long time and you need to restore it to a point using this method, then it will restore back to the beginning of your session. You will lose everything you've done since.

 If you are about to do something in the script that you may want to rollback, on the **Source Control** tab > in the **Restore Points** section, click **Create**.

- **Remove restore points when the application closes**

When you close PowerShell Studio, restore points will be removed.

## VersionRecall

---

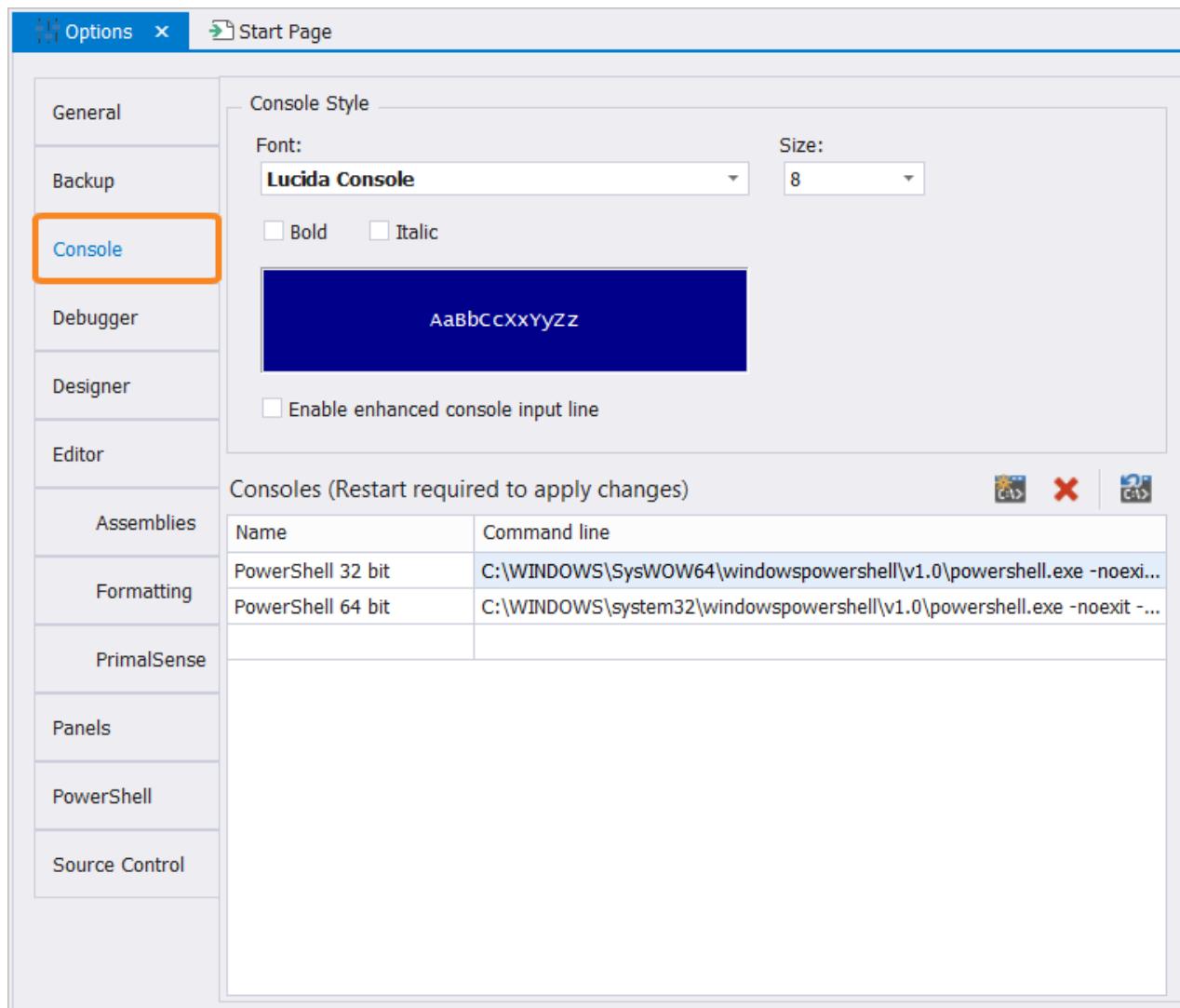
- **Automatically submit to VersionRecall repository when a file is closed**

VersionRecall is SAPIEN's version control system. This option allows you to save your script to VersionRecall when it closes.

 For information, visit the [VersionRecall product page](#).

## 14.4 Console

This topic covers the settings available in **Options > Console**. These settings allow you to configure the console style and manage embedded shells.



## In this section

- [Console Style](#) 339
- [Consoles](#)

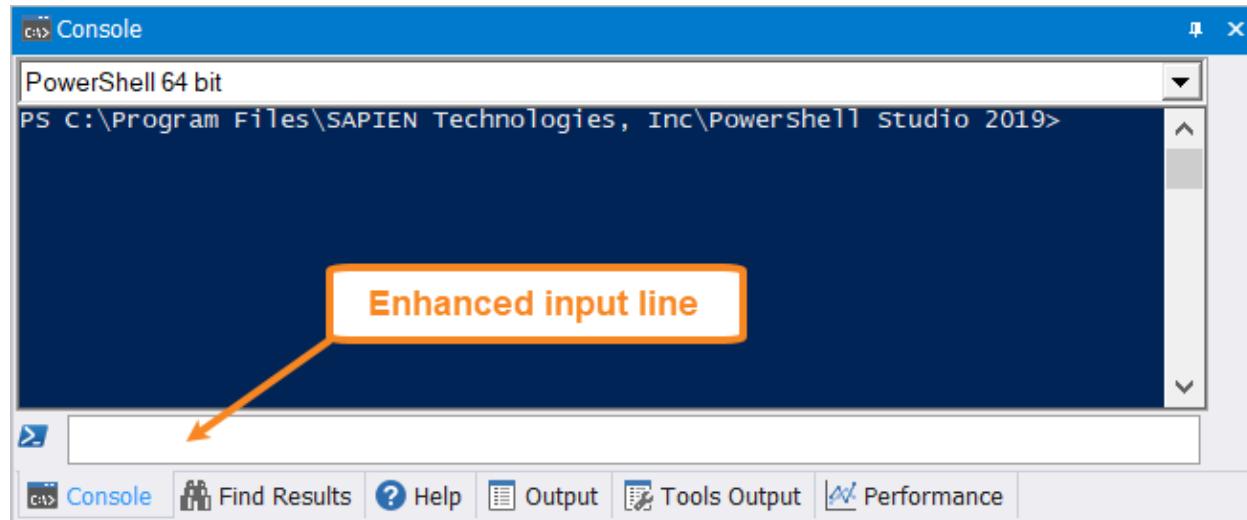
## Console Style

- **Font**  
Sets the Console font type.
- **Size**  
Sets the Console font size.
- **Bold**  
Sets the Console font as bold.
- **Italic**

Sets the Console font as italic.

- **Enable enhanced console input line**

Creates a separate input box to type in, instead of typing directly into the console.



- ⓘ To make your console style changes effective, click anywhere outside of the Options dialog or the Console panel.

## Consoles

On the first run, PowerShell Studio will attempt to detect Windows PowerShell and PowerShell Core and automatically add them to the Console list.

Consoles (Restart required to apply changes)	
Name	Command line
PowerShell 32 bit	C:\WINDOWS\SysWOW64\windowspowershell\v1.0\powershell.exe -noexit...
PowerShell 64 bit	C:\WINDOWS\system32\windowspowershell\v1.0\powershell.exe -noexit...
PowerShell Core 64 bit - 6	C:\Program Files\PowerShell\6\pwsh.exe -noexit -Command Remove-Mo...

There are three buttons at the top-right of the Consoles list:



From left to right:

- **Add Shell**

Navigate to the shell executable and then click **Open** to add a shell.

- **Remove**

Remove the selected shell.



• **Reset**

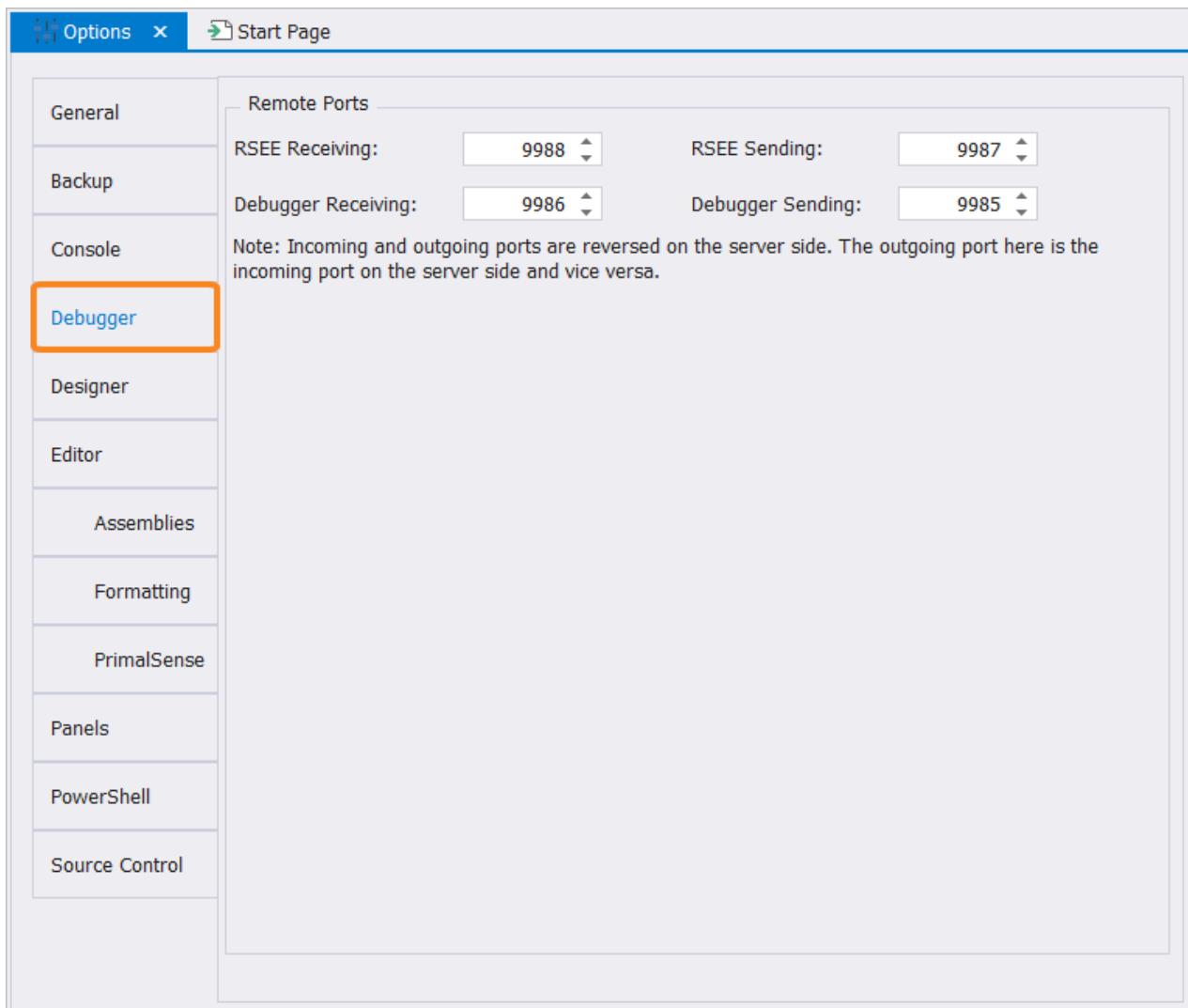
Restore the default PowerShell consoles.

 You must restart PowerShell Studio to apply the changes.

 If you install a new version of PowerShell Core, you will need to reset your consoles to see the new version in the console panel. If you don't want to reset any custom settings, then you will need to manually update the path to the PowerShell Core executable by clicking the three dots (...) to the right of the shell path.

## 14.5 Debugger

This topic covers the settings available in **Options > Debugger**.



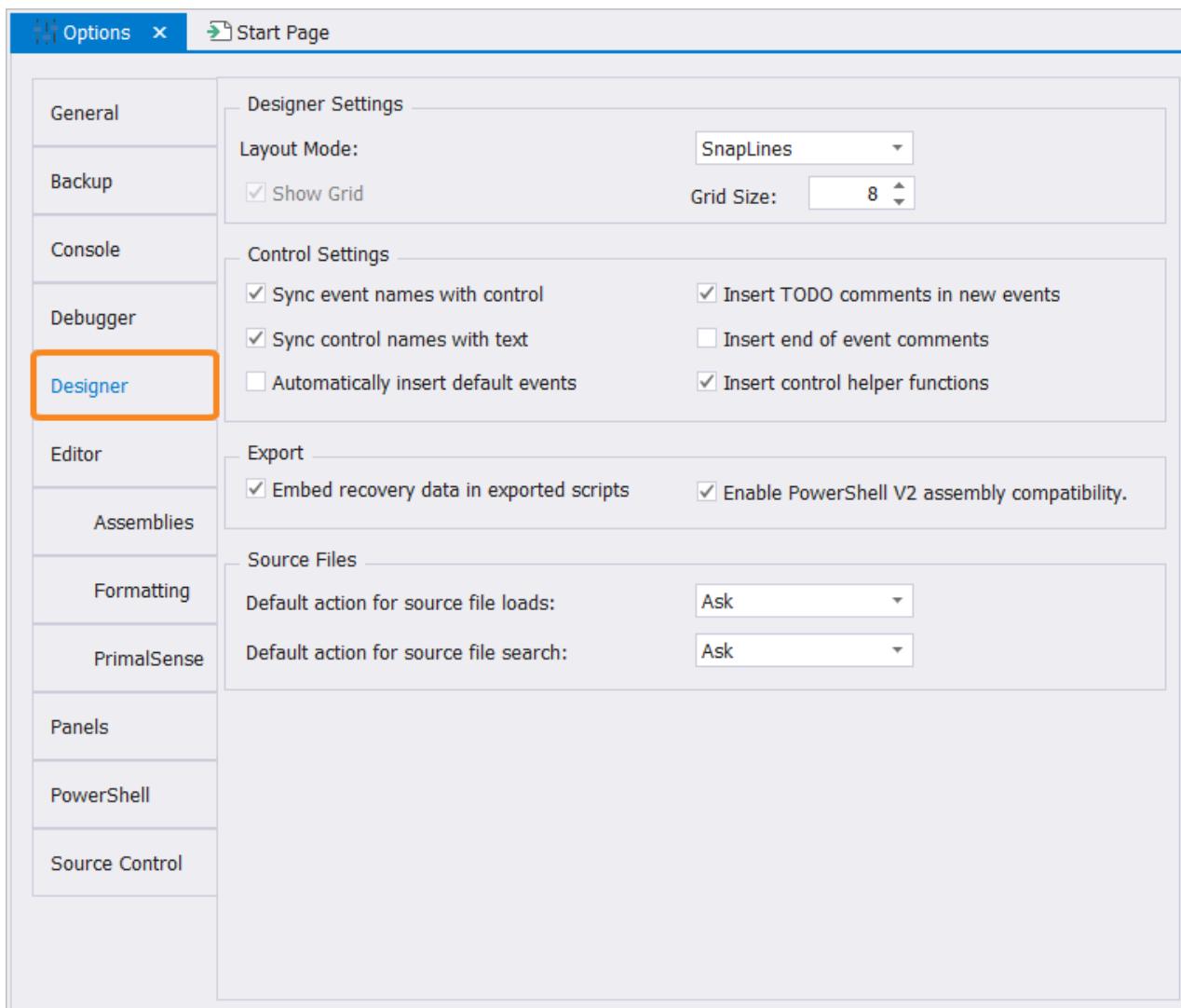
## Remote Ports

These settings configure the network ports that PowerShell Studio uses to connect to a remote installation of the [Remote Script Execution Engine](#) (RSEE). You must use the same port numbers on any computer that you want to support the remote execution. When deciding on which ports to use, it is important to consult your network and security teams, as they will be able to advise you which ports are safe to use and, if required, reconfigure any firewalls. Port numbers are specified in the registry on a machine that is running the RSEE service.

The key is HKEY\_LOCAL\_MACHINE\Software\Policies\SAPIEN. The Value name is **InPort** (for the incoming port) and **OutPort** (for the outgoing port). These values are most easily configured by means of a Group Policy Object (GPO), and we provide a template (ADM file) that can be imported into a GPO to configure RSEE.

## 14.6 Designer

This topic covers the settings available in Options > Designer.



### In this section

- [Designer Settings](#) 343
- [Control Settings](#) 344
- [Export](#) 344
- [Source Files](#)

## Designer Settings

- Layout Mode

Governs how controls are aligned when added to the forms designer.

- **SnapLines**

Allows you to precisely align controls. As you move a control around on a form, snap lines will appear that show how the control aligns with its neighbors.

- **SnapToGrid**

Aligns controls to a grid overlaid on the forms designer.

- **Show Grid**

Show or hide the grid.

- **Grid Size**

Increase or decrease the grid size.

## Control Settings

---

- **Sync event names with control**

Event handler names are generated using the pattern \$<control name>\_<event name>. This option ensures that when you rename a control its event handlers are also renamed.

- **Sync controlnames with text**

Event handler names are generated using the pattern \$<control name>\_<event name>. This option ensures that when you rename a control its event handlers are also renamed.

- **Automatically insert default events**

Many controls have a default event (click for a button, selected index changed for a ComboBox). This option will ensure that the default event is always connected to an event handler when you add a control to a form.

- **Insert TODO comments in events**

When enabled, PowerShell Studio will add a comment to each event handler reminding you to provide your code body.

- **Insert end of event comments**

Adds a comment at the end of each event handler.

- **Insert control helper functions**

Uncheck this option to prevent PowerShell Studio from inserting control helper functions into the script

## Export

---

- **Embed Recovery Data in Exported Scripts**

Enabling this option allows PowerShell Studio to add extra metadata to an exported script that allows it to recreate the original project or form (psf) that was used to create the export. The recovery data is stored in multi-line comment blocks in the exported script.

- **Enable PowerShell V2 assembly compatibility**

This setting determines the .NET assembly version used:

- **Enabled**

2.0 .NET assembly references will be used when generating GUI scripts, and a warning will be generated if any assembly does not support the .NET 2.0 Runtime.

- **Disabled**

4.0 .NET or later assemblies will be used, and a `#requires -Version` comment will be placed at the top of the generated script.

## Source Files

- **Default action for source file loads**

This setting controls how PowerShell Studio loads exported scripts.

There are three options:

- **Load**

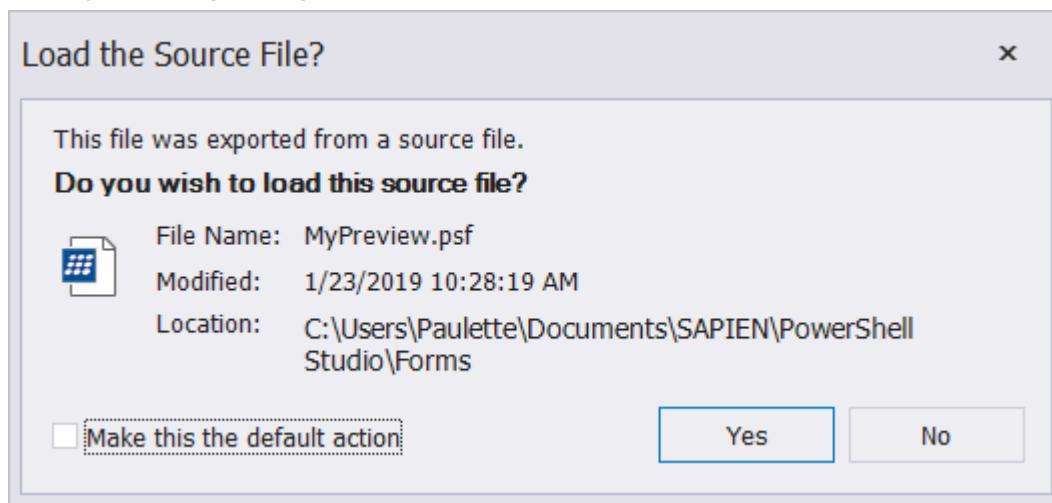
Reloads the original files that were used to create the exported script.

- **Ignore**

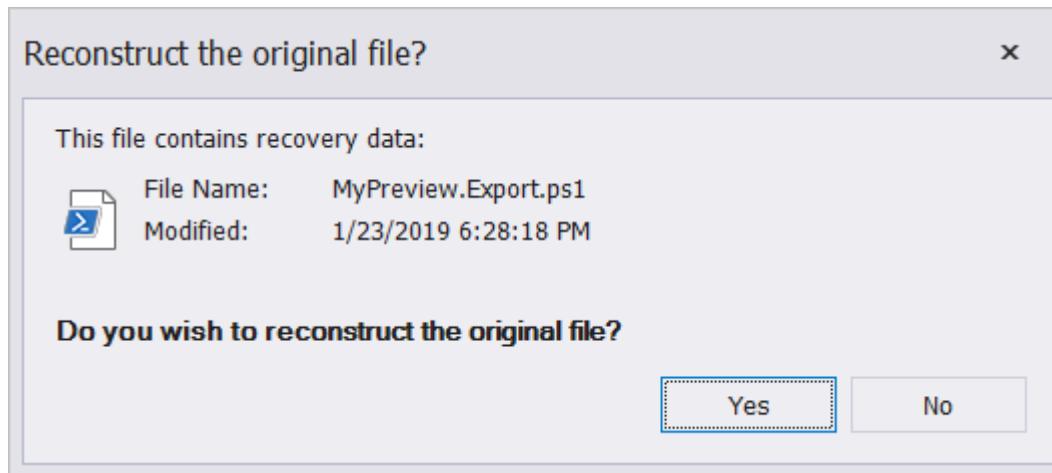
Loads the exported script as is.

- **Ask**

Displays a dialog asking if the exported file should be loaded.



- If the file contains recovery data you will also get a second prompt asking if you wish to use the recovery data to reconstruct the original files:



- **Default action for source file search**

This option controls how PowerShell Studio responds when it cannot find the original files for an exported script.

There are three options:

- **Search**

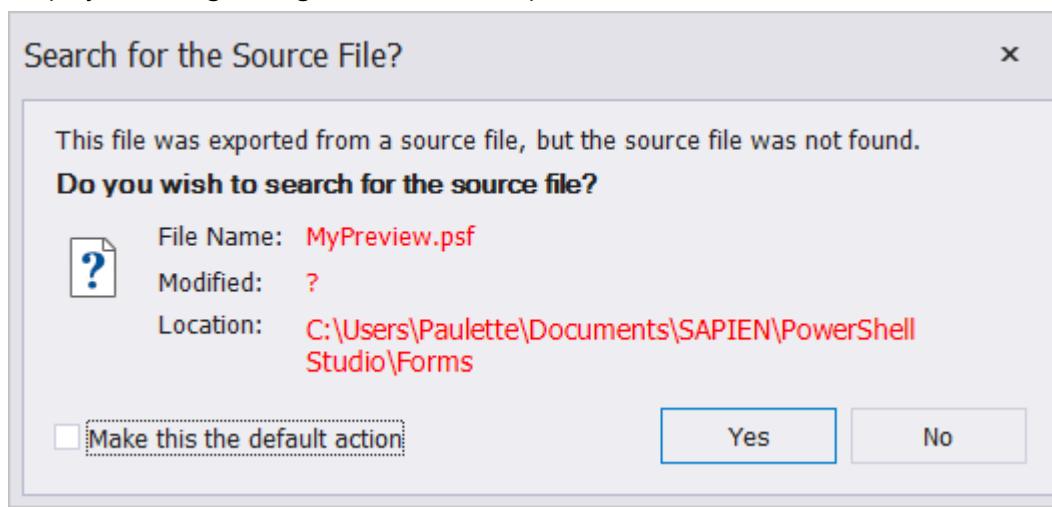
Tries to locate the missing files.

- **Never**

Does not attempt to locate the missing files.

- **Ask**

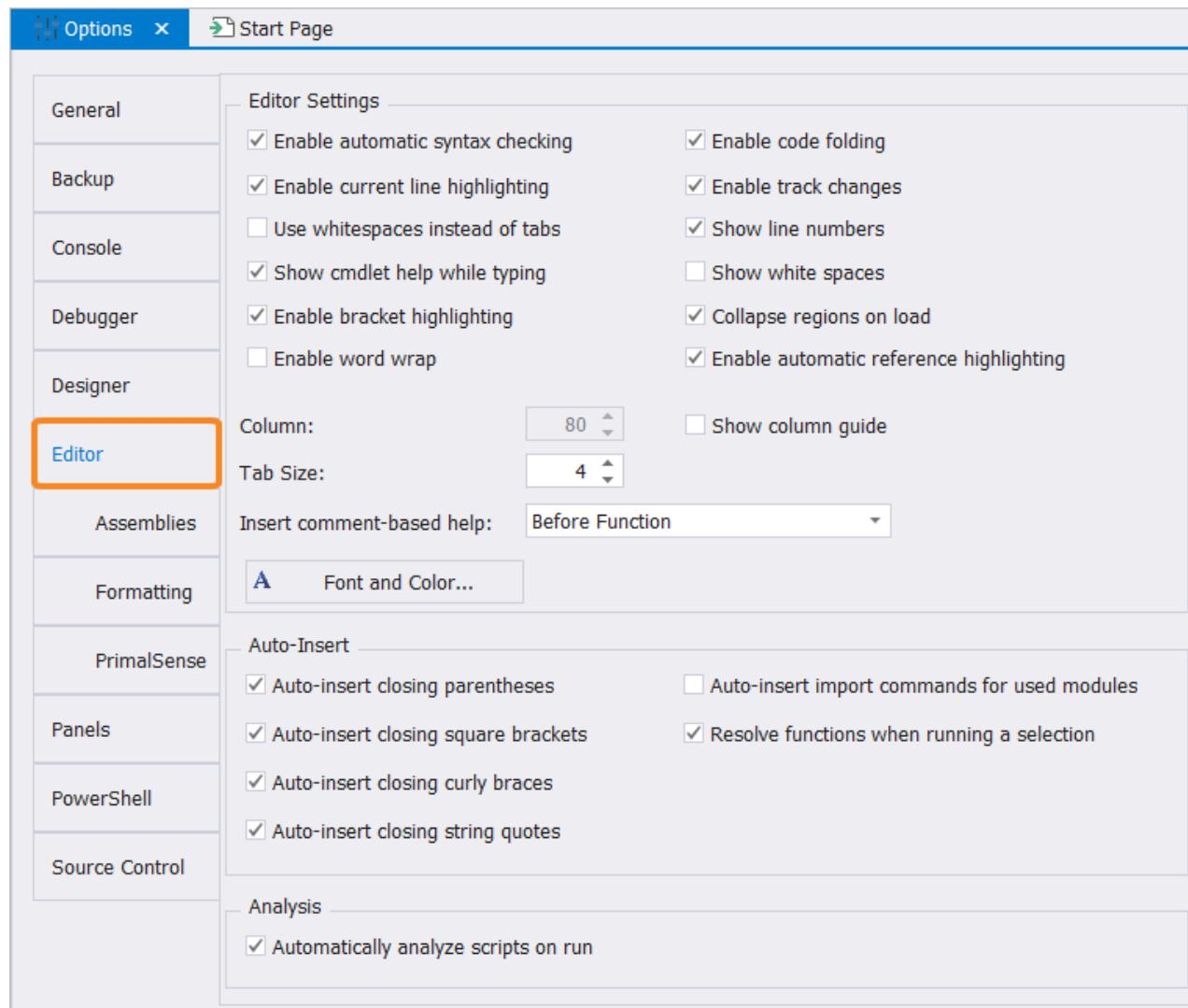
Displays a dialog asking the user how to proceed.



## 14.7 Editor

These topics cover the settings available in **Options > Editor**. These settings allow you to customize the appearance of the [script editor](#)<sup>46</sup>, designate external [assemblies](#)<sup>356</sup>, and also control code [formatting](#)<sup>357</sup> and [PrimalSense](#)<sup>366</sup> behavior.

These options customize the appearance of the code editor panel.



### In this section

- [Editor Settings](#)<sup>348</sup>
  - [Font and Color...](#)<sup>349</sup>
    - [Syntax Coloring](#)<sup>355</sup>
- [Auto-Insert](#)<sup>356</sup>
- [Analysis](#)

## Editor Settings

---

- **Enable automatic syntax checking**

Provides real-time syntax analysis as you type.

- **Enable current line highlighting**

Places a colored bar under the current line to provide a visual contrast.

- **Use whitespaces instead of tabs**

Inserts spaces rather than tabs into code when indenting.

- **Show cmdlet help while typing**

If you type a cmdlet it will appear in the Object Browser.

- **Enable bracket highlighting**

Highlights the opening and closing brackets when you click on either.

- **Enable word wrap**

Allows word wrap.

- **Enable code folding**

Allows script blocks, functions, comments, and regions to be collapsed to a single line in the code editor.

 For more information, see [Script Editor > Manipulating Regions](#) [49].

- **Enable track changes**

Annotates the source code with yellow and green bars to indicate changes made in the current editing session.

- **Show line numbers**

Displays line numbers in the left margin of the code editor.

- **Show white spaces**

Show white spaces that trail behind the back-tick at the end of a line.

- **Collapse regions on load**

Collapse the *Include* nodes when a script is loaded (configure *Include* nodes at Home > Edit > Regions > Include).

- **Enable automatic reference highlighting**

Automatically highlight the references of the current selection or the object under the caret. Automatically highlight the relevant object-based references when caret position is changed.

- **Column**

Specifies the column position where the column guide should be displayed. *Show column guide* must be enabled.

- **Show column guide**

Displays a vertical line at a particular column.

- **Tab size**

Specifies the tab size.

- **Insert comment-based help**

There are two options:

- **Before Function**

Inserts the comment right before the function declaration.

- **Inside Function**

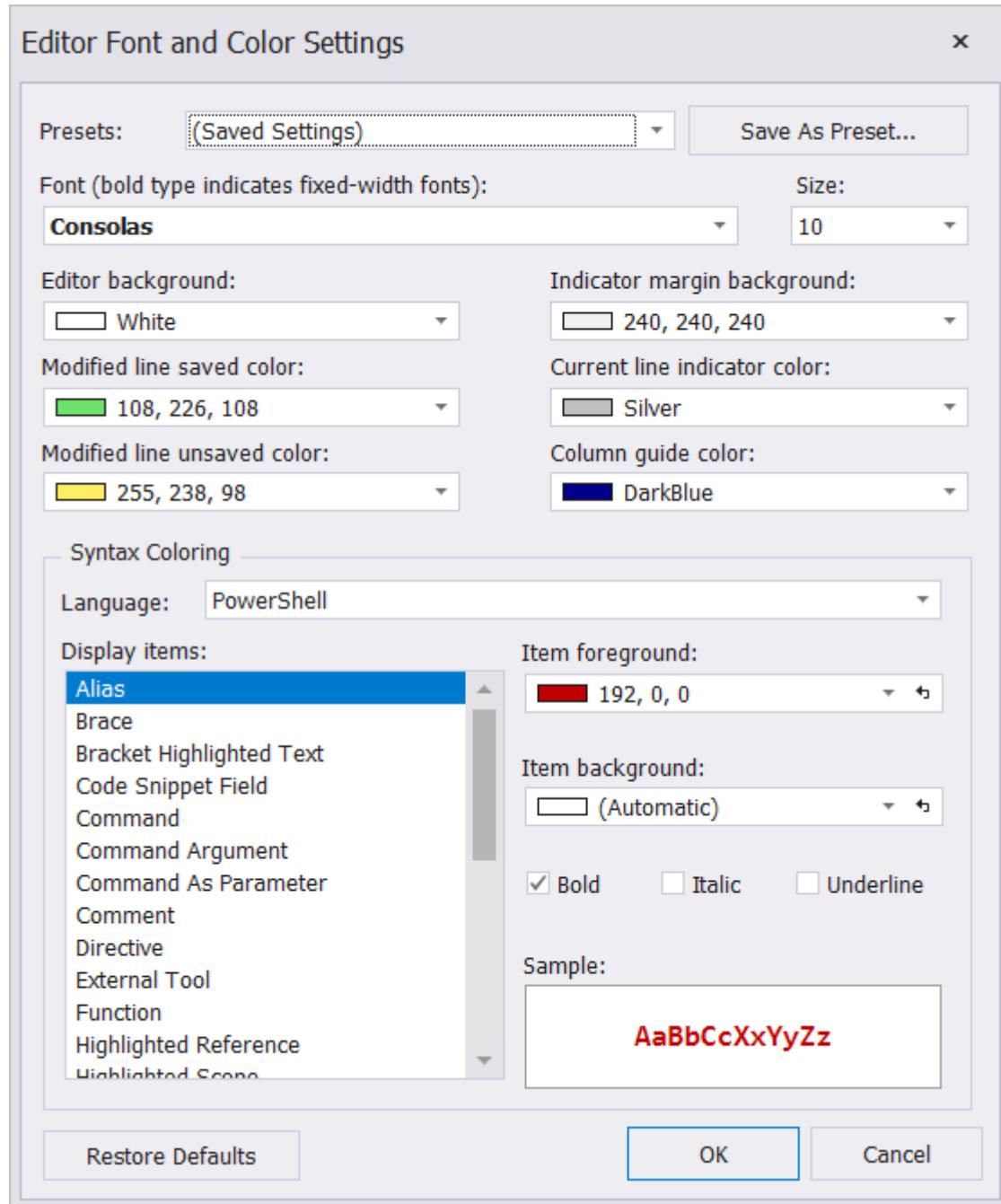
Inserts the comment within the function declaration, before the parameter block.

 For more information, see [Script Editor > Comment-Based Help](#) 

## Font and Color...

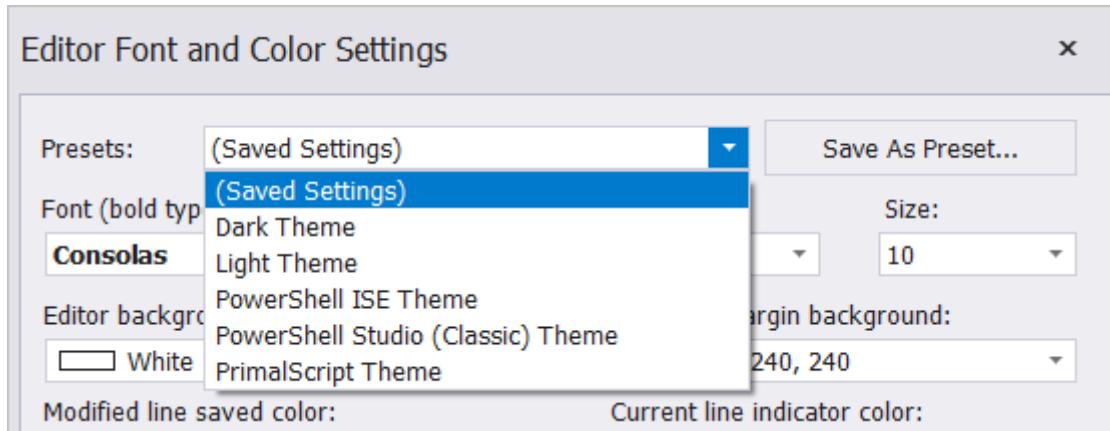
---

The Editor Font and Color Settings dialog allows you to customize the font and coloring of PowerShell Studio's editor, including syntax coloring for supported languages.



- **Presets**

Predefined font and color themes are available from the drop-down list at the top of the dialog.



There are six default preset options:

- **(Saved Settings)**  
Restores the settings to the last saved state.
- **Dark Theme**  
Changes the color scheme and font to a dark colored theme.
- **Light Theme**  
Changes the color scheme and font to a light colored theme.
- **PowerShell ISE Theme**  
Changes the color scheme and font to match the default settings of the Microsoft PowerShell ISE.
- **PowerShell Studio (Classic) Theme**  
Changes the color scheme to the default PowerShell Studio coloring and font.
- **PrimalScript Theme**  
Changes the color scheme to the default PrimalScript coloring and font.

## To select a preset

1. Select an option from the Presets drop-down; PowerShell Studio will update the font and color settings to the predefined preset.
2. Click **OK** to apply the selected theme, and PowerShell Studio will use the new coloring.

### • Save As Preset...

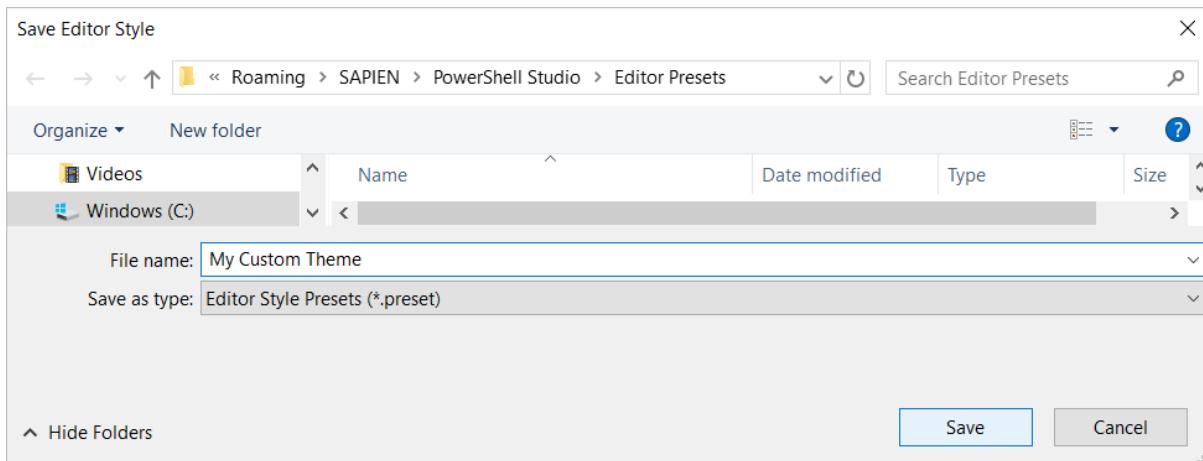
Saves your custom font and color settings as a preset.

## To save a preset

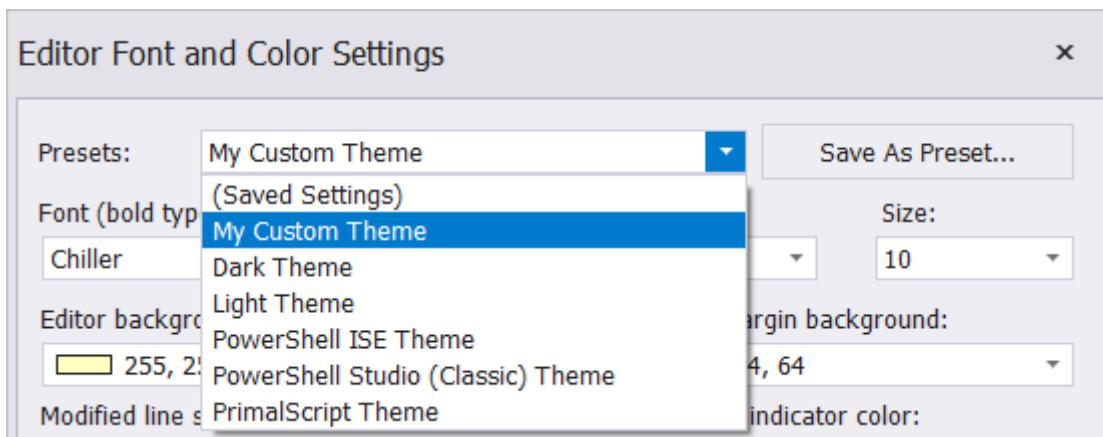
1. Configure the font and color settings, and then click the **Save As Preset...** button:

**Save As Preset...**

2. In the Save Editor Style dialog, enter a name for the preset, and then click **Save**:



The custom preset will be displayed in the Presets drop-down list:



## To import a preset

Copy the preset file (\*.preset) to the following user specific folder:

C:\Users\<username>\AppData\Roaming\SAPIEN\PowerShell Studio\Editor Presets\

The imported preset will appear in the drop-down list the next time you edit the font and colors.

 Share your custom presets with colleagues by sending them your preset file (\*.preset).

### • Font

Sets the editor font. All bolded fonts are fixed-width fonts.

### • Size

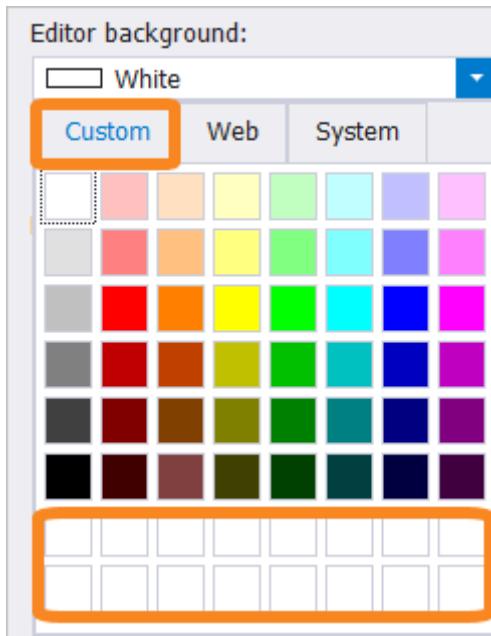
Sets the editor font size.

### • Editor background

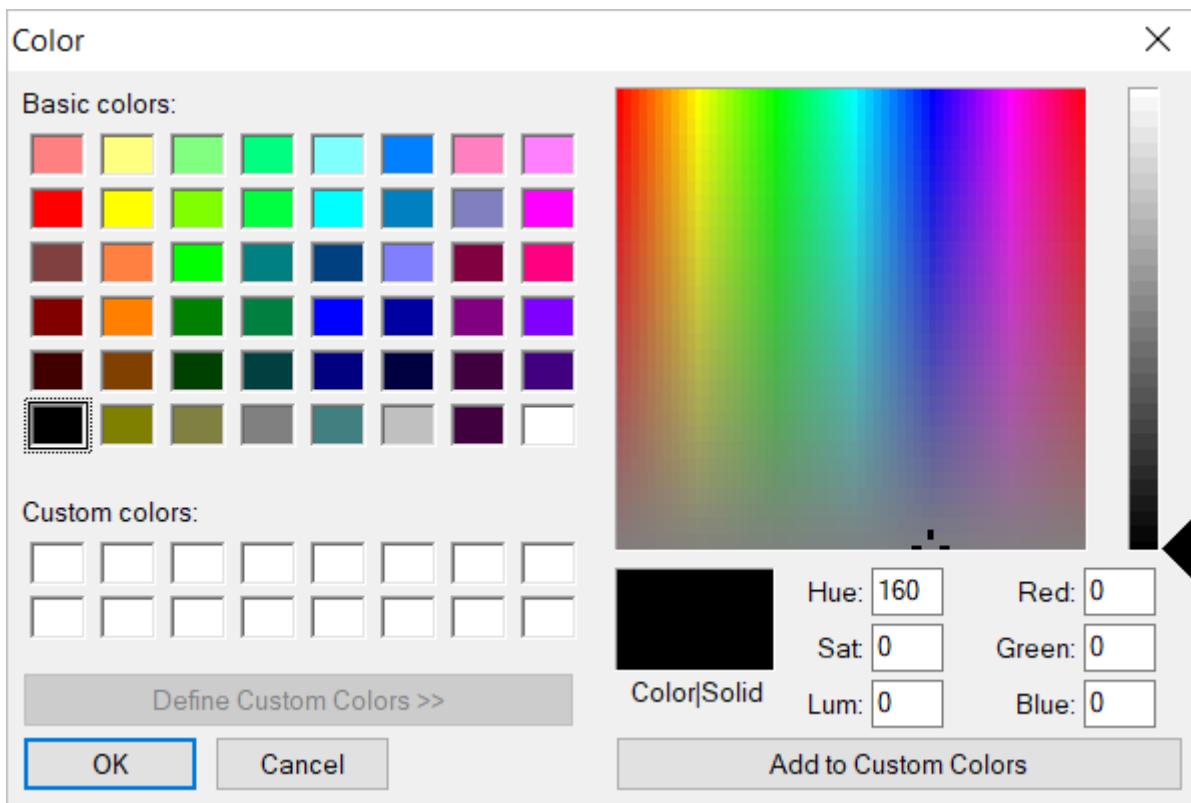
Sets the color for the editor background.

## To define a custom color

1. Click the color picker drop-down and select the **Custom** tab.
2. Right-click on any empty color space:



The Color editor allows you to manually enter RGB values or use sliders to customize and select the desired color:



- **Modified line saved color**

Sets the color for modified lines that have been saved.

```

6
7
8 function Load-MyChart
9 {
10

```

An orange arrow points to the vertical margin line on the left side of the code editor, indicating the color of the indicator margin for saved modified lines.

- **Modified line unsaved color**

Sets the color for modified lines that have not been saved.

```

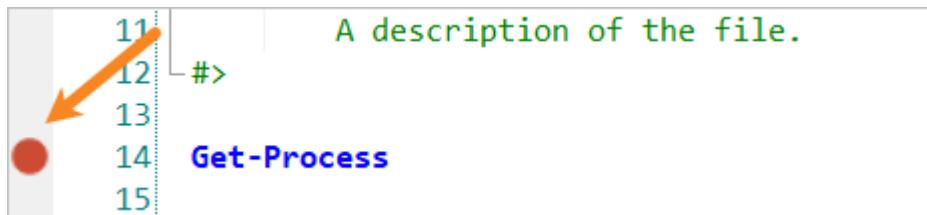
6
7
8 function Load-MyChart
9 {
10

```

An orange arrow points to the vertical margin line on the left side of the code editor, indicating the color of the indicator margin for unsaved modified lines.

- **Indicator margin background**

Designates the background color of the indicator margin where the breakpoints, tracepoints, and bookmarks are located.



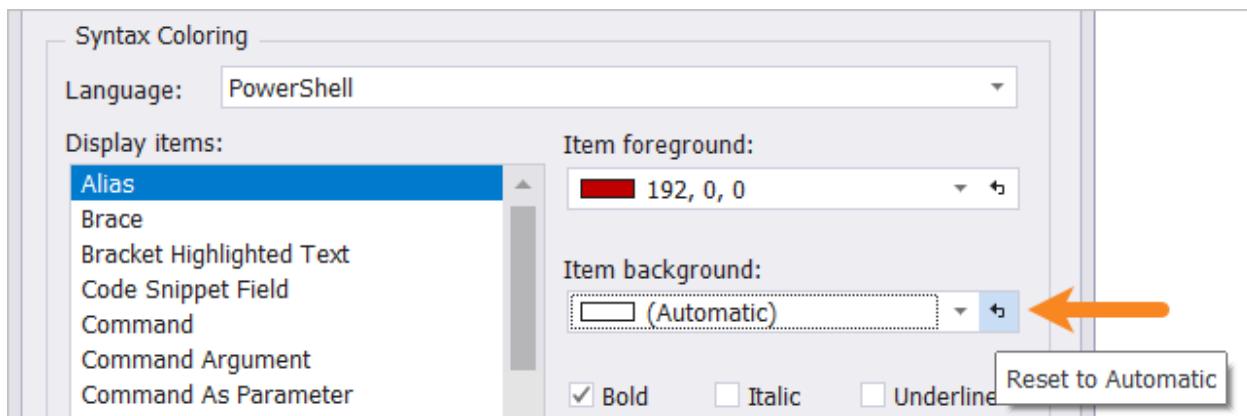
- **Current line indicator color**  
Sets the color for the current line indicator.
- **Column guide color**  
Sets the color for the column guide.

## Syntax Coloring

Designates theme coloring and style for supported languages and file types. This feature is very helpful when working with a variety of file types; for example, when working with module projects.

- **Language**  
Select the language or file type.
- **Display items**  
Items available that can have their displayed color or style changed.
- **Item foreground**  
Sets the color for the selected item foreground. The arrow to the right of the color selector drop-down will reset the color to *(Automatic)*.
- **Item background**  
Sets the color for the selected item background.

>To set the background transparent for the selected item, click the **Reset to Automatic** arrow to the right of the color selector drop-down:



- **Bold**  
Sets the selected item to display as **bold**.
- **Italic**

Sets the selected item to display as *italicized*.

- **Underline**

Sets the selected item to display as underlined.

- **Restore Defaults**

Resets the language colors back to the default color settings (light theme).

## Auto-Insert

---

- **Auto-insert closing parenthesis**

Auto-completes parenthesis while you type.

- **Auto-insert closing square brackets**

Auto-completes square brackets while you type.

- **Auto-insert closing curly braces**

Auto-completes curly braces while you type.

- **Auto-insert closing string quotes**

Auto-completes string quotes while you type.

- **Auto-insert import commands for used modules**

When you type a cmdlet that is part of a module that isn't imported in the script, PowerShell Studio will automatically insert the Import-Module statement.

- **Resolve functions when running a selection**

Bypasses the requirement to include the function definition in the selection when using the **Run Selection** command. In a Project context, PowerShell Studio will also automatically resolve functions defined in other project files.

 This option only applies to **Run Selection** and not to **Run Selection in Console**.

## Analysis

---

- **Automatically analyze scripts on run**

Trigger PSScriptAnalyzer every time you run / debug a PowerShell script or a Project. The analysis is displayed in the Tools Output panel.

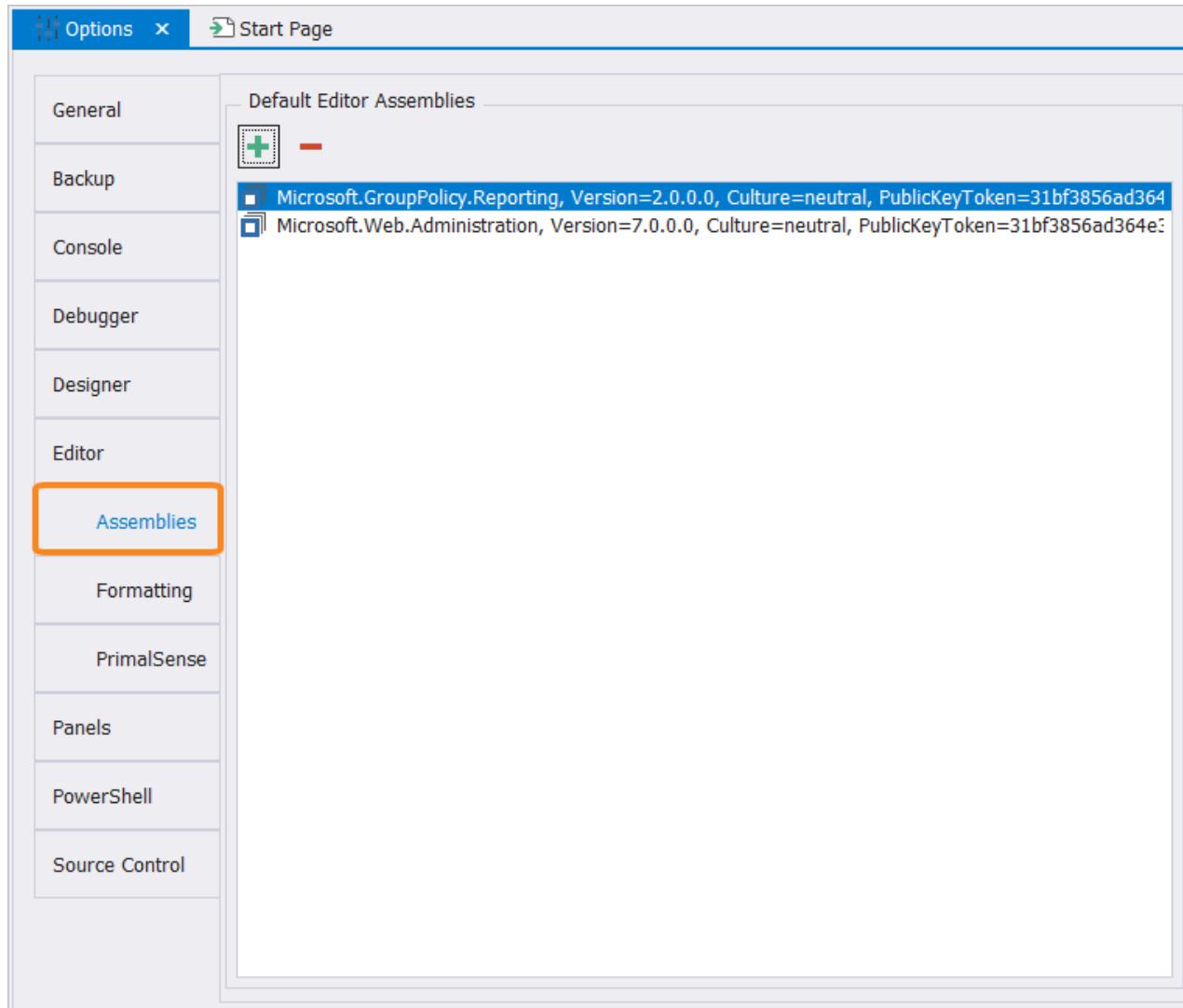
 Requires installation of the PSScriptAnalyzer module.

### 14.7.1 Assemblies

This topic covers the settings available in Options > Editor > Assemblies.

The Default Editor Assemblies page lists external assemblies that you explicitly designate to load every time PowerShell Studio is started.

- i** PowerShell Studio provides PrimalSense support for base .NET assemblies such as *mscorlib*, *system.data*, etc., and also ensures that the *System.Windows.Form* assembly is loaded (when running GUI scripts)—without explicitly listing the assemblies in the Default Editor Assemblies page.

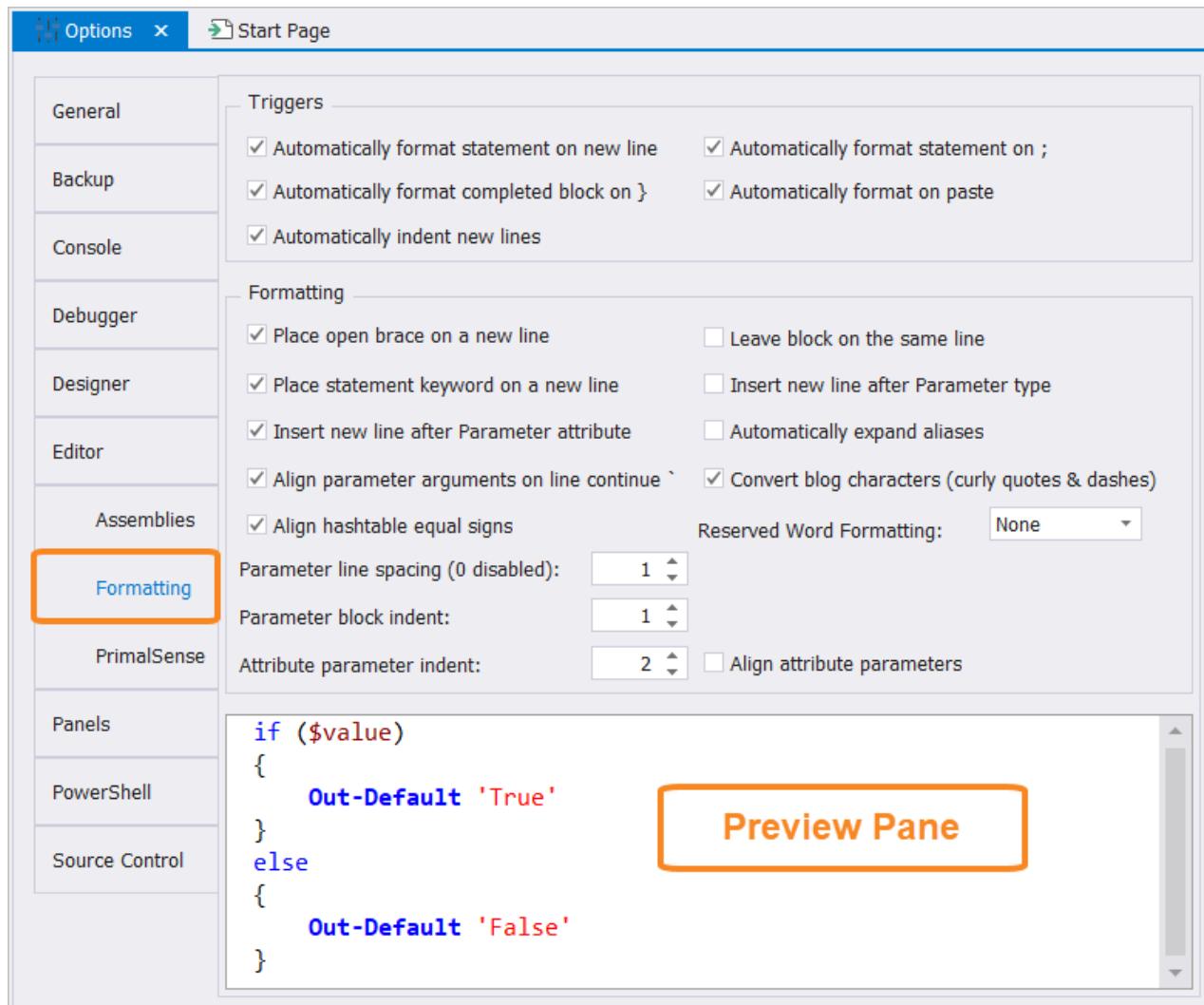


- i** GUI psf files created with older versions of PowerShell Studio (2017 and earlier) will retain the old assemblies list.

## 14.7.2 Formatting

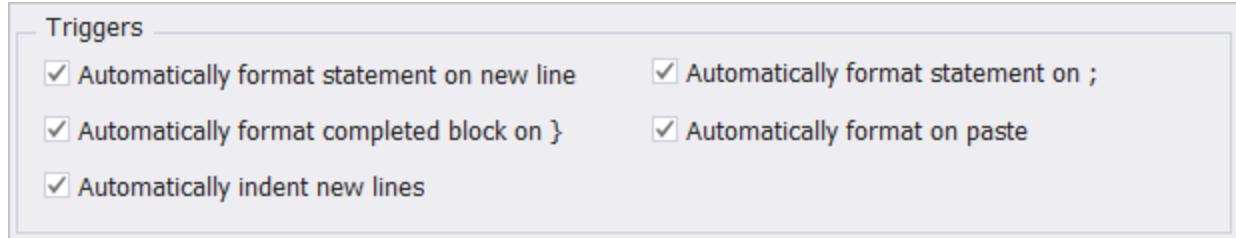
This topic covers the settings available in Options > Editor > Formatting.

PowerShell Studio's code formatting is customizable to fit your needs:



## Triggers

The first few options serve as triggers for auto-formatting. These options tell PowerShell Studio when to format your script:



The following triggers can be enabled:

- **Automatically format statement on new line**

Formats the line when you press <Enter>. You might not notice anything if the text is already formatted.

- **Automatically format completed block on }**

Formats a code block when the closing curly bracket is typed.

- **Automatically indent new lines**

Automatically indents the appropriate tab depth when inserting a new line by pressing <Enter>.

- **Automatically format statement on ;**

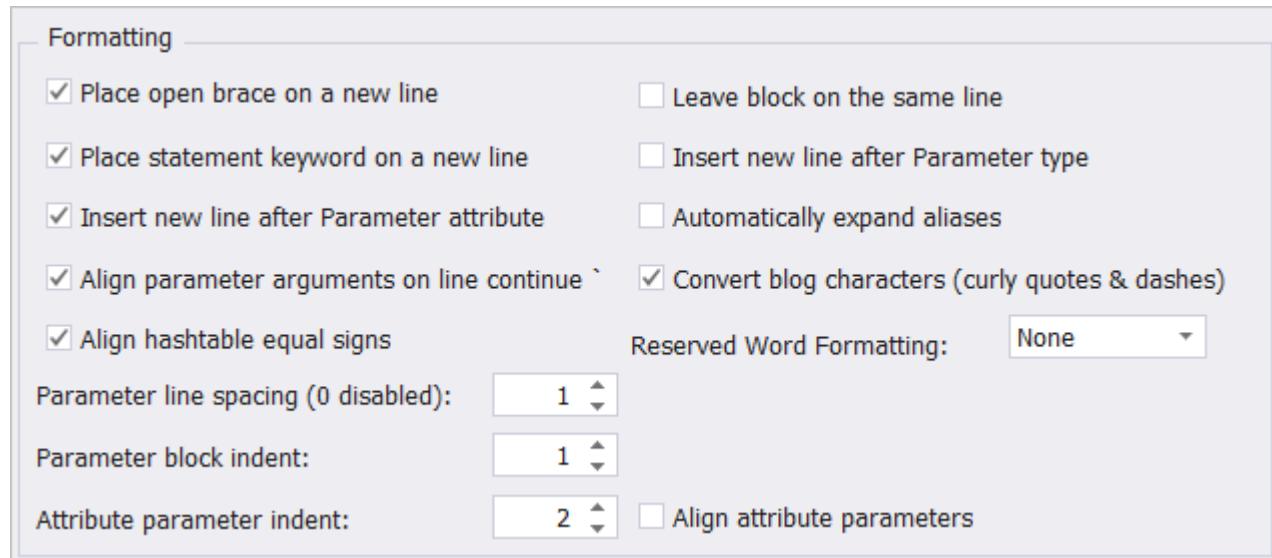
Formats the statement when the semi-colon is typed.

- **Automatically format on paste**

Formats code when it is pasted into the script.

## Formatting

The second set of options allows you to customize the formatting rules of PowerShell Studio:



**i** Most settings shown below will be altered in the preview pane when changed, allowing you to see what effect they have on your script.

The following formatting rules can be configured:

- **Place open brace on a new line**

Places new open curly braces on a new line.

*Enabled*

```
if ($value)
{
    Out-Default 'True'
}
else
{
    Out-Default 'False'
}
```

*Disabled*

```
if ($value) {
    Out-Default 'True'
}
else {
    Out-Default 'False'
}
```

- Place statement keyword on a new line

Places each statement key word on a new line, such as the *if* and the *else* in an *if/else* statement.

*Enabled* (*Place open brace on a new line = Disabled*)

```
if ($value) {
    Out-Default 'True'
}
else {
    Out-Default 'False'
}
```

*Disabled* (*Place open brace on a new line = Disabled*)

```
if ($value) {
    Out-Default 'True'
} else {
    Out-Default 'False'
}
```

- Insert new line after Parameter attribute

Places each parameter attribute on its own line.

*Enabled*

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

## Disabled

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )][ValidateNotNullOrEmpty()][String]$Param1,
    [ValidateRange(1, 10)][int]$Param2
)
```

- Align parameter arguments on line continue

When parameter arguments are continued to a new line using the back tick line continuation character, they are aligned on the same column.

## Enabled

```
Get-Service -ComputerName 'Server'
    -Name 'WinRM'
    -Verbose
```

## Disabled

```
Get-Service -ComputerName 'Server'
    -Name 'WinRM'
    -Verbose
```

- Align hashtable equal signs

Aligns hashtable equal signs on separate lines.

## Enabled

```
$parameters = @{
    Path      = 'C:\ProgramData'
    Filter    = '*.*'
    Recurse   = $true
}
```

## Disabled

```
$parameters = @{
    Path = 'C:\ProgramData'
    Filter = '*.*'
    Recurse = $true
}
```

- Only equalizes spacing for the first key value pair on any given line.

```
$ hashtable = @{
    Frequency = 'Weekly'; DaysOfWeek = 'Monday', 'Wednesday', 'Friday';
    At         = '23:00'; Interval = 2
}
```

- Leave block on the same line

The script block stays on the same line.

## Enabled (Place open brace on a new line = Disabled)

```
if ($value) { Out-Default 'True' }
else { Out-Default 'False' }
```

## Disabled (Place open brace on a new line = Enabled)

```
if ($value) {
    Out-Default 'True'
}
else {
    Out-Default 'False'
}
```

- Insert new line after Parameter type

Inserts a new line after the type of the parameter.

## Enabled

```
Param (
    [Parameter(Mandatory = $true,
               Position = 0,
               HelpMessage = 'Parameter 1'
               )]
    [ValidateNotNullOrEmpty()]
    [String]
    $Param1,
```

## Disabled

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

- **Automatically expand aliases**

Expands command and parameter aliases.

## Enabled

```
Get-Process -Name 'PowerShell Studio'
```

## Disabled

```
ps -Name 'PowerShell Studio'
```

- **Convert blog characters (curly quotes & dashes)**

Replaces curly quotes (blog quotes) with straight quotes, and replaces extended dashes with en dashes.

## Enabled

```
Out-Default "Curley Quotes" -Transcript
```

## Disabled

```
Out-Default "Curley Quotes" -Transcript
```

- **Reserved Word Formatting**

Formats reserved word (keyword) character casing.

- **None**

Leaves the reserved word casing as is.

- **Upper**

Uses all upper-case characters:

IF

IFELSE

WHILE

- **Lower**

Uses all lower-case characters:

```
if  
ifelse  
while
```

- **Camel**

Uses all camel case characters:

```
If  
IfElse  
While
```

- **Parameter line spacing (0 disabled)**

Governs the line spacing between parameters. If set to a value greater than zero, the specified number of new lines will be inserted after the parameter.

*Enabled (spacing = 2)*

```
[Parameter](Mandatory = $true,  
           Position = 0,  
           HelpMessage = 'Parameter 1'  
           )]  
[ValidateNotNullOrEmpty()]  
[String]$Param1,  
  
[ValidateRange(1, 10)]
```

*Disabled (spacing = 0)*

```
[Parameter](Mandatory = $true,  
           Position = 0,  
           HelpMessage = 'Parameter 1'  
           )]  
[ValidateNotNullOrEmpty()]  
[String]$Param1,  
[ValidateRange(1, 10)]  
[int]$Param2
```

- **Parameter block indent**

The number of tab spaces a parameter block will be indented.

*Enabled (indent = 1)*

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

*Disabled (indent = 0)*

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

- **Attribute parameter indent**

The number of tab spaces an attribute parameter will be indented. This setting is used when 'Align attribute parameters' is disabled.

*Enabled (indent = 1)*

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

*Disabled (indent = 0)*

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

- Align attribute parameters

Vertically aligns a parameter's attributes declarations. When disabled, the 'Attribute parameter indent' value is used to indent the subsequent lines of attributes.

*Enabled*

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

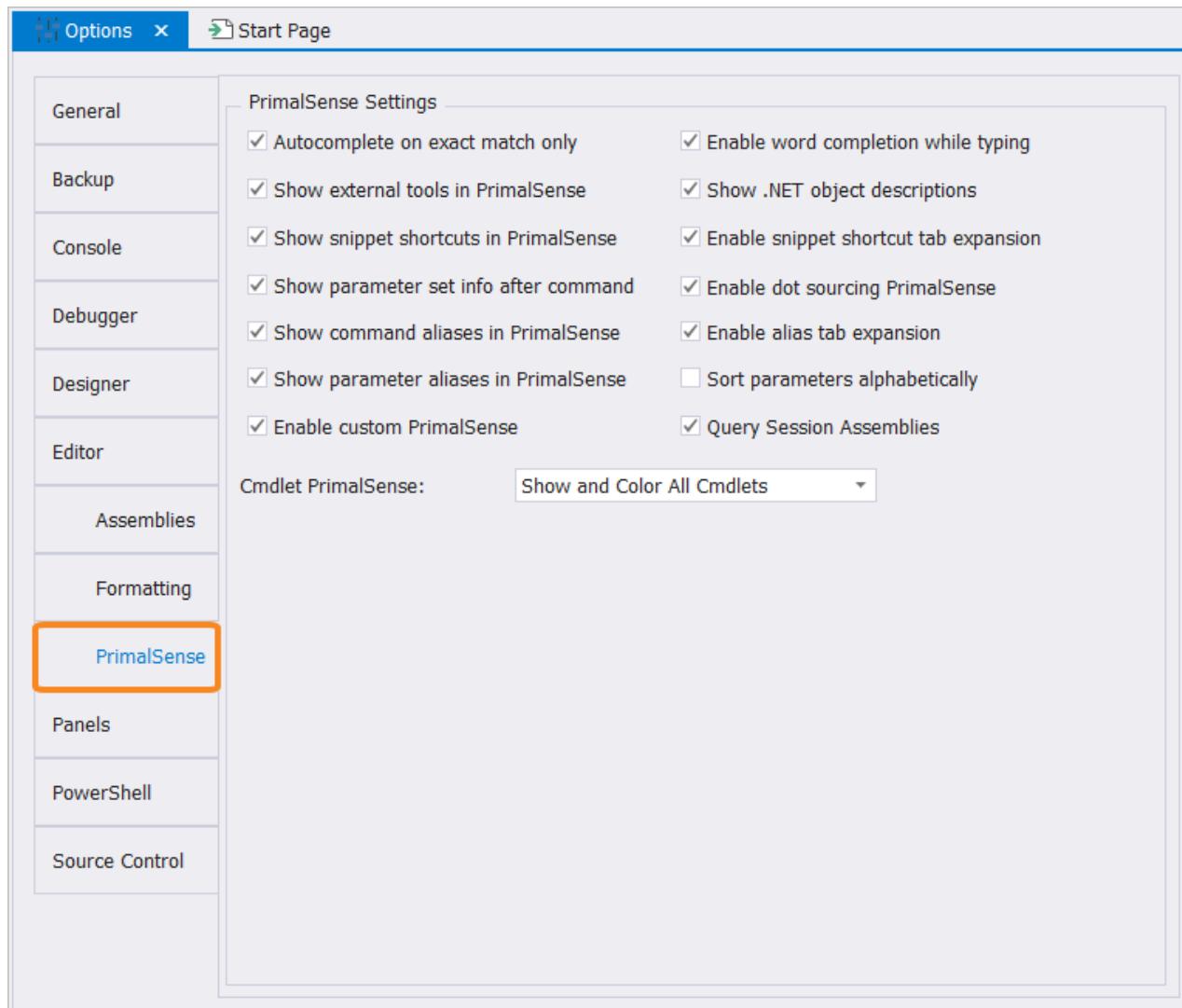
*Disabled (Attribute parameter indent = 2)*

```
Param (
    [Parameter(Mandatory = $true,
        Position = 0,
        HelpMessage = 'Parameter 1'
    )]
    [ValidateNotNullOrEmpty()]
    [String]$Param1,
    [ValidateRange(1, 10)]
```

## 14.7.3 PrimalSense™

This topic covers the settings available in Options > Editor > PrimalSense™.

These options allow you to configure the behavior of PrimalSense.



## PrimalSense Settings

- **Autocomplete on exact match only**

Governs PrimalSense's selection and matching behavior.

- **Enabled**

Double-clicking or using the <Tab> key will auto-complete the item if the search expression is a complete match (fully selected).

- **Disabled**

Using the <Space> key will trigger auto-completion of a partial match (partially selected).

- **Show external tools in PrimalSense**

Displays command line tools within PrimalSense's autocomplete list (the tools are cached).

- **Show snippet shortcuts in PrimalSense**

While typing, PrimalSense will list snippet shortcuts along with their command names.

- **Show parameter set info after command**

When typing a space after a command, a pop-up window is displayed allowing you to cycle through the command's parameter sets.

- **Show command aliases in PrimalSense**

When typing a command, PrimalSense will suggest aliases.

- **Show parameter aliases in PrimalSense**

When you type or specify the parameter in a command, PrimalSense will suggest aliases.

- **Enable custom PrimalSense**

Allows customized PrimalSense derived from a static list or from a dynamically created list using a PowerShell script.

- **Enable word completion while typing**

Fills in the missing characters on partially typed words (after typing three or more characters). The drop-down list may provide one or multiple possible completions.

- **Show .NET object descriptions**

PrimalSense will display help text when you hover over an object.

- **Enable snippet shortcut tab expansion**

Pressing tab at the end of a snippet will expand it into the full name.

- **Enable dot sourcing PrimalSense**

When you dot source a file in PowerShell Studio, the file will be automatically loaded and parsed to provide PrimalSense and coloring for functions contained in the file.

- **Enable alias tab expansion**

Pressing tab at the end of an alias will expand it into the full name.

- **Sort parameters alphabetically**

Disable this to display common command parameters at the end of the PrimalSense list.

- **Query Session Assemblies**

Loads the debug session's assemblies to provide syntax coloring and PrimalSense, when at a breakpoint.

- **Cmdlet PrimalSense**

There are three options:

- **Show and Color All Cmdlets**

Enables PrimalSense to show all cmdlets whether they are loaded or not.

- **Show Cmdlets (Active Modules Only)**

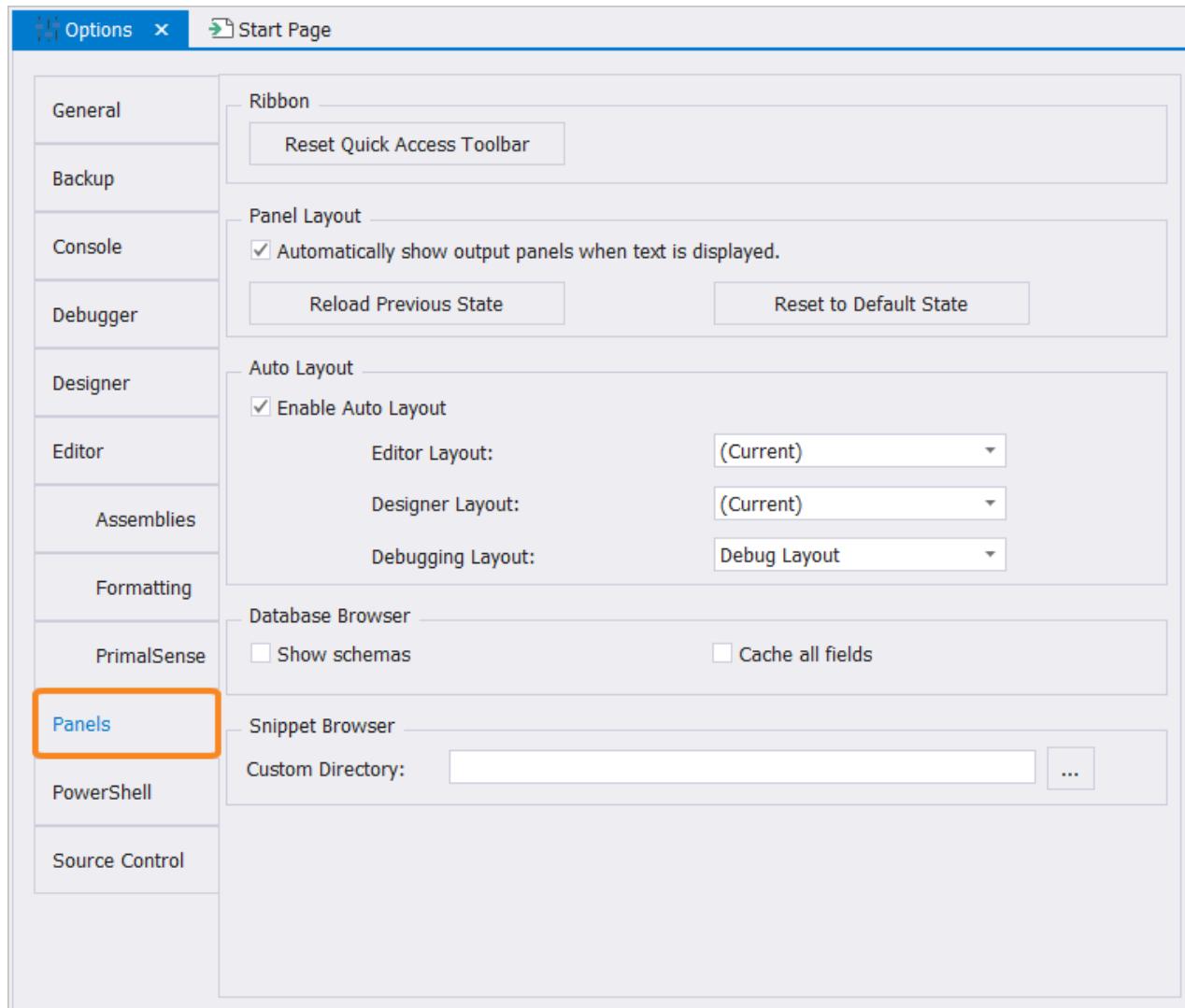
PrimalSense will only show cmdlets that are part of the current project.

- **Show All Cmdlets (PrimalSense Only)**

PrimalSense will offer suggestions from all of the PowerShell modules on your system.

## 14.8 Panels

This topic covers the settings available in Options > Panels.



## In this section

- [Ribbon](#) [369]
- [Panel Layout](#) [370]
- [Auto Layout](#) [370]
- [Database Browser](#) [370]
- [Snippet Browser](#)

## Ribbon

- **Reset Quick Access Toolbar**

Resets the Quick Access Toolbar to its default state.

## Panel Layout

- **Automatically show output panels when text is displayed**

Display output panels when text is displayed.

 When a panel has new output, the panel's tab will highlight when it is docked or auto-hidden.

- **Reload Previous State**

Resets PowerShell Studio to the same state as the last time you opened it.

- **Reset to Default State**

Configures PowerShell Studio to its default panel layouts.

## Auto Layout

- **Enable Auto Layout**

Allows PowerShell Studio to arrange and display panels based on the current context. For example, when debugging, all of the debug related panels will be visible and all other panels will be hidden.

- **Editor Layout**

Designates your preferred layout for editing code. Choosing (*Current*) will keep the layout you are currently using.

- **Designer Layout**

Designates your preferred layout for designing forms. Choosing (*Current*) will keep the layout you are currently using.

- **Debugging Layout**

Designates your preferred layout for debugging. Choosing (*Current*) will keep the layout you are currently using.

## Database Browser

- **Show schemas**

This setting configures how the [Object Browser](#)<sup>214</sup> displays schema information from SQL server:

- **Disabled**

The Object Browser will ignore schema information and display database tables, stored procedures, etc. in a simple flat list.

- **Enabled**

The Object Browser will display database objects in their respective schemas.

- **Cache all fields**

This setting will cache everything within the [Database Browser](#)<sup>218</sup> for quick loading of recently used information. Checking this option will increase the time for the Database Browser to cache and load the database.

## Snippet Browser

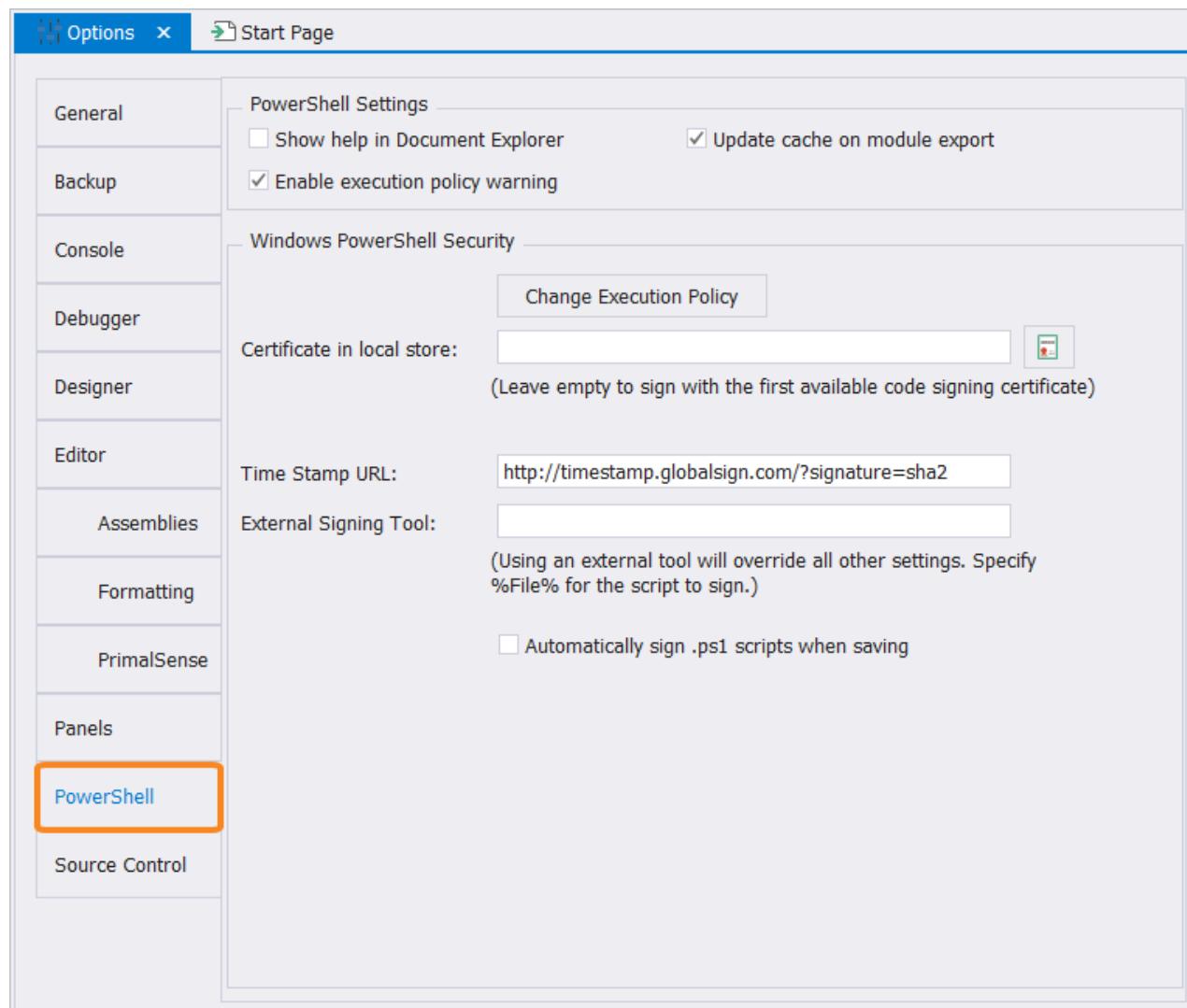
- Custom Directory

Adds an extra folder to the [Snippets panel](#) [245]

👉 Provide a network path here to create a shared snippet repository.

## 14.9 PowerShell

This topic covers the settings available in Options > PowerShell.



## PowerShell Settings

---

- **Show help in Document Explorer**

Displays help in Document Explorer instead of in the Help panel. Turning on this option disables context-sensitive help in the Help panel.

- **Enable execution policy warning**

PowerShell Studio will warn you if the current PowerShell execution policy is set to Restricted, and will help you to change it if required.

- **Update cache on module export**

Updates the stored cache when exporting modules.

## Windows PowerShell Security

---

- **Change Execution Policy**

Allows you to reconfigure the execution policy for both 32-bit and 64-bit shells from a simple GUI interface.

- **Certificate in local store**

The name of the certificate that PowerShell Studio will use for code signing.

- **Password**

The password required to access the certificate stored in PFX format.

- **Time Stamp URL**

Used to add a timestamp to the signature block in a script. This provides an extra level of security, enabling PowerShell to determine if the certificates used to sign a script were valid when the script was signed.

- **External Signing Tool**

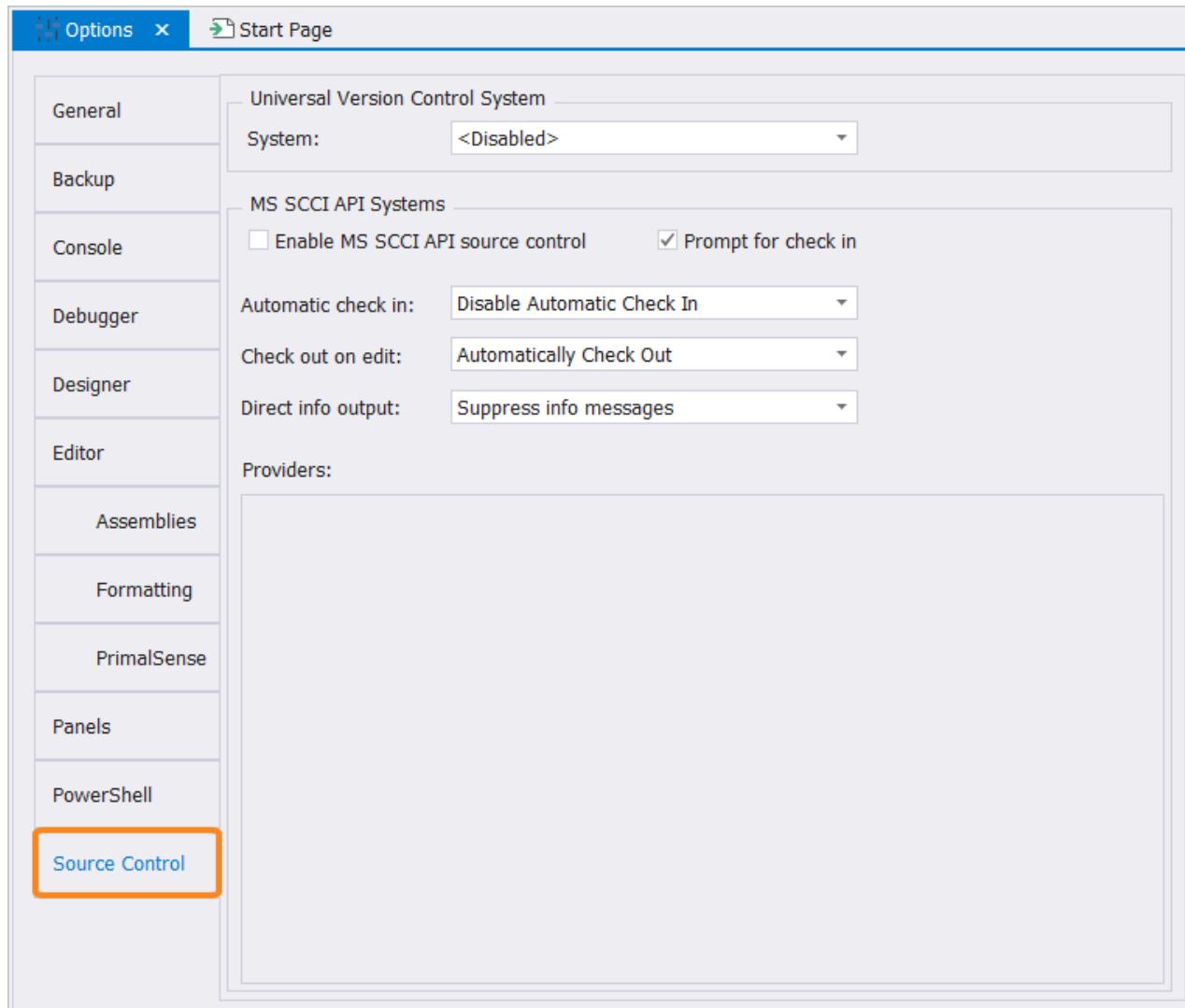
Provides the name of an alternative code signing tool.

- ***Automatically sign .ps1 script when saving***

Ensures that all scripts are signed.

## 14.10 Source Control

This topic covers the settings available in Options > Source Control, which allow you to configure source control systems and providers.



## Universal Version Control System

The Universal Version Control system allows configuration of any source control provider with command line tools.

- **System**

The source control system / provider.

- **Disable**

Disables the Universal Version Control feature.

- **Git**

Enables version control support using Git.

## MS SCCI API Systems

- **Enable MS SCCI API source control**

Allows PowerShell Studio to use source control if it is available.

- **Prompt for check In**

This option works in conjunction with Automatic check in:

<i>Prompt for Check In</i>	<i>Automatic Check In</i>	<i>Behavior</i>
Checked	Check In when a file is closed.	User is prompted to check in files.
Unchecked	Check In when a file is closed.	PowerShell Studio automatically checks in files.
Any value	Disable Automatic Check In.	User must manually check files in.

- **Automatic check in**

This controls Check In behavior and has the following two options:

- **Disable Automatic Check In**

Let's you decide when files should be checked in.

- **Check In when a file is closed**

Automates the check in process.

- **Check out on edit**

This controls Check Out behavior and has the following three options:

- **Automatically Check Out**

PowerShell Studio will check a file out as soon as you edit the file.

- **Prompt For Check Out**

PowerShell Studio will display a file chooser dialog in which you can choose the files you wish to check out. The file you are currently attempting to edit will automatically be selected.

- **Direct info output**

This controls where output messages are directed, and has the following three options:

- **SUPPRESS info messages**

Hides all messages.

- **Write to output panel**

Messages are displayed in the Output Panel.

- **Display in message box**

Messages are displayed in a pop-up dialog.

- **Providers**

Displays the list of available source control providers and allows the user to switch between them.

## 15 Remote Script Execution Engine

The Remote Script Execution Engine™ (RSEE™) is an enterprise-level remote script execution environment.

### RSEE Overview

RSEE consists of two components: The client, which is built into PrimalScript and PowerShell Studio, and a remote service that must be deployed to each computer where you will remotely run scripts. RSEE is capable of deploying a script from within PrimalScript and PowerShell Studio, out to remote computers where the script is executed, and bringing the scripts' output and results back to PrimalScript or PowerShell Studio for your review.

RSEE is a complex tool and it interacts closely with Windows' security subsystems. RSEE is recommended for use only by experienced Windows administrators who fully understand service deployment and management, cross-computer security and authentication and, in the case of domain environments, Group Policy objects and Active Directory administration. Apart from the guidelines in this manual, SAPIEN Technologies cannot assist you with security issues caused by improper configuration nor can we assist with Active Directory, Group Policy, or local computer configuration tasks.

RSEE is designed only for Windows Script Host (WSH) scripts in VBS (VBScript) or JS (JScript) files. It is not designed for other WSH scripts (including WSFs) nor is it designed for scripts written in other languages (such as batch, KiXtart, and so forth).

### RSEE Deployment

RSEE's service component is packaged in a **Microsoft Windows Installer (MSI)** file and is suitable for deployment via Group Policy. You can also manually install it on individual machines. Keep in mind that, once installed, the service needs to be started in order to be useful. This will occur automatically after restarting the computer on which the service is installed (the service is set to start automatically by default).

After deploying the service, there are a number of configuration steps that you must take in order to properly configure RSEE in your environment.

### Identity

RSEE installs, by default, to log in under the privileged LocalSystem account. This may be sufficient for your purposes. However, when deploying scripts in PrimalScript and PowerShell Studio, be sure not to specify any credentials in the Launch dialog box. Also be advised that the LocalSystem account may not be able to execute some scripts, depending on their security requirements.

We recommend that you configure the RSEE service to run under a user account that has administrative privileges on the local computer. In a workgroup environment this would be a local account, and we recommend creating the same local account (with the same password) on all of your com-

puters, for consistency. In a domain environment, we recommend creating a single domain account which has local administrative rights on all computers in the domain, and using this account to run the RSEE service. Whenever the RSEE service is running under a user account, you must specify that account (and its password) when deploying scripts in PrimalScript.

When using RSEE, you have the option to specify the credentials under which the script should execute. Generally speaking, you need to provide the same credentials that the RSEE service is using to log on.

## TCP Port

---

The RSEE service defaults to using TCP port 9987 for incoming connections, and TCP port 9988 for outgoing connections. It is your responsibility to ensure that any local firewalls will permit incoming traffic on this port. Keep in mind that the Windows Firewall (Windows XP SP 2 and later, and Windows Server 2003 SP 1 and later) can be centrally configured via a domain Group Policy object.

### To specify a different port

---

- You can specify a different port via the registry key `HKEY_LOCAL_MACHINE\Software\Policies\SAPIEN`. The Value name is `InPort` (for the incoming port) and `OutPort` (for the outgoing port). Note that these values are most easily configured by means of a Group Policy object (GPO), and we provide a template (ADM file) that can be imported into a GPO to configure RSEE.

The RSEE service *and* both PrimalScript and PowerShell Studio (as the RSEE client) utilize `InPort` and `OutPort`. The service listens to `InPort` for incoming connections and uses `OutPort` to send script output back to the client. The client reverses this: scripts are sent via `InPort` and results are received on `OutPort`. The registry key above configures these ports for both clients and the service.

## Domain Tips

---

While manually configuring a few computers in a workgroup is not a hardship, manually configuring an entire domain of computers can be burdensome. An Active Directory domain environment provides a number of capabilities for centralizing and automating this configuration, however. While this section is not intended as a comprehensive tutorial in Active Directory (we recommend that you consult an experienced Active Directory administrator or the appropriate documentation if you need more assistance), the following tips should help you configure RSEE more easily:

- **Create a domain account**

Name this account something like "RSEEUser" and provide it with a strong password per your organization's password policies.

- **Deploy the RSEE service**

This can be done by means of a Group Policy object (GPO) linked to the appropriate levels in the domain. The RSEE service defaults to running under the LocalSystem account and it defaults to port 9987. The service's MSI is located in the RSEE folder under your PrimalScript Enterprise install-

ation folder.

- **Make the RSEE service account a local Administrator**

You can do this in a Group Policy object (GPO). Browse to Computer Configuration > Security Settings > Restricted Groups. Add a group ("Administrators") and then add your RSEE domain account (and any other appropriate accounts) to the group.

- **Configure the RSEE service**

You need to configure the RSEE service to log on with the user account (and password) you created. This can either be done manually or using a script. The book Windows Administrator's Automation Toolkit, for example, contains a script that can set the logon account and password used by services running on multiple computers. Utilities like Service Explorer ([www.scriptlogic.com](http://www.scriptlogic.com)) can perform the same task.

- **Select the TCP port**

We provide a Group Policy object (GPO) administrative template (ADM file) that you can import into a GPO and use to centrally configure the TCP port used by the RSEE service. This ADM file is located in the RSEE folder under your PrimalScript Enterprise installation folder.

## Using RSEE

---

RSEE now supports Powershell. To deploy the current script (only VBS and JS files are currently supported) to one or more remote computers that have the RSEE service installed, click the RSEE button on the Script toolbar, or select Run Script on Remote Computer from the Script menu.

RSEE performs a quick scan of your script to look for commands that might create a graphical user element such as the VBScript MsgBox() function. If it finds any of these functions, it displays a warning message. Keep in mind that scripts will not normally be able to interact with the desktop environment on remote computers, meaning there would be no way for someone to respond to graphical elements such as MsgBox() or InputBox(). As a result, these elements can cause the script to "hang" and stop responding. RSEE does not perform an exhaustive check for graphical elements; it is your responsibility to ensure they're not used in your scripts. RSEE will allow you to continue with graphical elements because you may have configured the RSEE service to interact with the desktop of the remote computer. It's your decision.

### RSEE Launch dialog

---

The Launch dialog lists the computers where your script will be deployed. Note that the Launch dialog always preloads a default list of computer names at startup. Here's what you can do:

- Click Launch to run the script on the computers which have a checkmark next to their name.
- Set or clear the checkbox next to one or more computer names. You can leave names in the list but clearing their checkbox will prevent RSEE from attempting to run the script on them.
- Click Close to close the Launch dialog. If you've changed the list of computer names, you'll be prompted to save your changes.

- Use Load List and Save List to load an alternate list of computer names (from a text file) or save the current list to a text file. By default, PrimalScript will look for a text file called Default.clt in the \SAPIEN\RSEE Lists folder under your Documents folder. You will need to create the file yourself if you want a pre-loaded list when you launch RSEE.
- Use Select All and Unselect All to set or clear the checkbox next to all computer names currently in the list.
- Select a computer name and click Remove to remove it from the list.
- Type a computer name (must be resolvable to an IP address by your computer) or IP address and click + to add that computer to the list.
- Specify a username (user ID) and password. These will be used to run the script on the remote computer, and should generally match the username that the remote RSEE service is using to log in. Note: if the username you specify is a local account on the remote computer(s), then just type the username. If the username is a domain account, specify the name in the format user@domain. The older domain\user format is not supported.

When you click Launch, RSEE will execute the script on the remote computer(s). Any output produced by the script will be displayed in the Output pane within PrimalScript or PowerShell Studio. Note that the message "Socket connection failed" indicates that RSEE was unable to connect to the RSEE service on a specified computer (either because the computer is not connected to the network, has a firewall blocking the RSEE service ports, or the RSEE service is not installed).

RSEE deploys scripts asynchronously. That is, RSEE sends the scripts out to the remote computers you've selected and then displays whatever results come back. If your scripts produce no output then you won't see any results in PrimalScript or PowerShell Studio.

It's possible for the RSEE service on a remote computer to run into a problem (particularly security-related ones) that it can't report back; in these instances, it will seem to you (looking at PrimalScript or PowerShell Studio) as if nothing has happened. Whenever possible, your scripts should incorporate error-checking and -trapping, and should produce appropriate output so that you get some results back if the script executes correctly.

Note that RSEE cannot be used to deploy a script for later execution. If you need to schedule a script to execute on a remote computer at a particular time, use Windows' built-in Task Scheduler instead of RSEE. You can even write a script utilizing the SCHTASKS.EXE command line tool that creates remote scheduled tasks on multiple computers.

Also note that, if an **Output** pane is already open in PrimalScript or PowerShell Studio, RSEE will utilize it rather than creating a new one. You will need to manually select the tab to view any RSEE results or error messages.

## RSEE Restrictions

In order to bring the output of remote scripts back to your computer, the remote RSEE service cap-

tures the standard command-line output of your scripts. That means any script output must be created using the WScript.Echo method. **Do not** use graphical user interface functions such as MsgBox() or InputBox(). Because the RSEE service doesn't interact with the desktop, nobody will ever see these functions' dialog boxes and the script will hang.

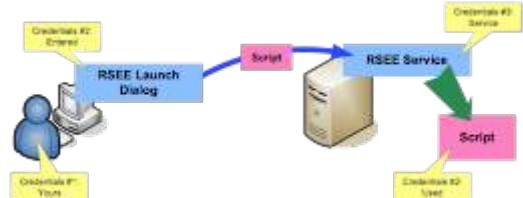
It is possible, if the RSEE service is running under the LocalSystem account, to configure Windows to allow the service to interact with the desktop. You may wish to experiment with this configuration, but it is not a recommended configuration because of the usual security restrictions on the LocalSystem account.

Also avoid any object methods—such as WScript.Popup—that create graphical elements.

Any objects referenced by a script must be installed, registered, and available on the remote machine where RSEE executes the script.

At this time, RSEE can only be used to execute Windows Script Host scripts. RSEE explicitly launches scripts under CScript.exe which must be available on the remote computers.

Most other restrictions in RSEE are actually Windows security restrictions. When the RSEE service launches, it does so using the credentials you configure in Windows' service manager. When the RSEE service receives a script, it creates a brand-new process using whatever credentials you enter into the RSEE Launch dialog. The following figure illustrates this process and the three sets of credentials involved:



*RSEE Credentials and Execution Process*

Always bear in mind that your scripts execute under the security credentials you provide (Credentials #2 in the diagram). This process does require your attention, as several things can go wrong if you're not careful:

- If you specify credentials in the Launch dialog (#2 in the diagram) that the RSEE service account (#3 in the diagram) doesn't have permission to use in a new process launch, then script execution will fail.

Practically speaking, the credentials you provide in the Launch dialog (#2 in the diagram) need to be the same as the credentials the RSEE service uses to log in (#3 in the diagram).

- If the RSEE service account (#3 in the diagram) doesn't have appropriate rights (including "Log on as a service"), then the RSEE service will not be able to start.
- If your script tries to do something that the Launch credentials (#2 in the diagram) don't have permission to do—such as log into a database or access a file share—then you'll receive an error. Depending on the exact situation, this may or may not be communicated back to you in Prim-

alScript or PowerShell Studio.

- If your script tries to perform an illegal operation—such as specifying alternate credentials in a WMI connection (which is illegal because the script is executing locally on the remote machine, and local connections to WMI aren't allowed to use alternate credentials)—you'll receive an error. Again, depending on the exact circumstances, this error may or may not be fed back to you in PrimalScript or PowerShell Studio.

These and other similar situations are not problems with RSEE; they are inherent conditions of the Windows operating system and its security subsystems. Whenever you encounter an error with RSEE, bear these conditions in mind and think about the possible security ramifications of what your script is trying to do.

## RSEE Notes

---

RSEE encrypts scripts during transmission to help keep them secure.

RSEE does not implement any sort of IP filtering capability (which might, for example, allow you to ensure that only your computer can utilize RSEE on remote servers). Instead, we recommend using Windows' own built-in IP filtering (available as part of Windows' IPSec features). Using this filtering, you can ensure that only specified IP addresses are allowed to communicate on the TCP ports used by the RSEE service, thus restricting who can contact that service and utilize RSEE.

## 16 Reference

This section provides an overview of the SAPIEN Updates tool, and lists the keyboard shortcuts available in PowerShell Studio.

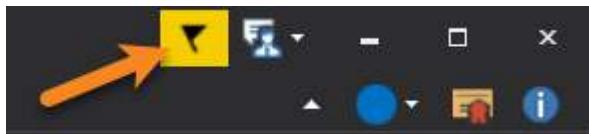
### 16.1 SAPIEN Updates

We are continually updating our software, both to remove bugs and to add and improve product features. We recommend always staying current with the most recent versions to ensure that you are taking advantage of the latest features, functionality, and product stability.

Every SAPIEN product has a built-in update tool—**SAPIEN Updates**—which will check for updates on all current activations and unexpired trial versions of our products. Available product updates are indicated in the SAPIEN Updates tool and also in the [Notifications dialog](#) [382] (see below).

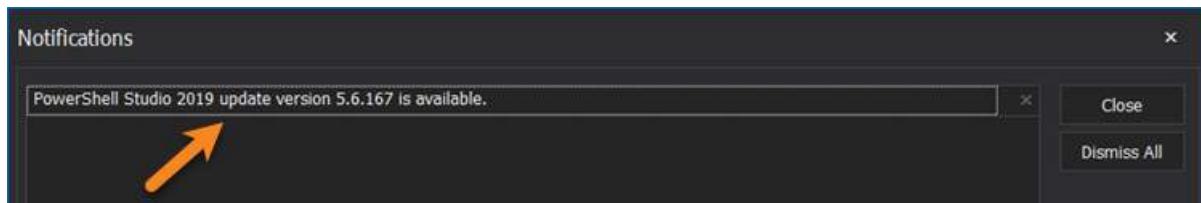
#### SAPIEN Notifications

SAPIEN products provide automatic notifications when there is a software update available, or when your maintenance is about to expire. Notifications are indicated by a 'flag' icon in the top-right of the program window:



#### How to view SAPIEN notifications

- Click the notification flag icon above the ribbon to open the Notifications dialog:



- If a product update is available, click the update notification to open the SAPIEN Updates tool.
- Click the X button to dismiss individual notifications or select **Dismiss All**. Dismissed notifications will not be shown again.

#### SAPIEN Updates - Tool Overview

The SAPIEN Updates tool indicates when an update is available for any SAPIEN program installed on your computer.

- i** To minimize the impact on your system, the tool does not run during Windows startup or continuously in the system tray.

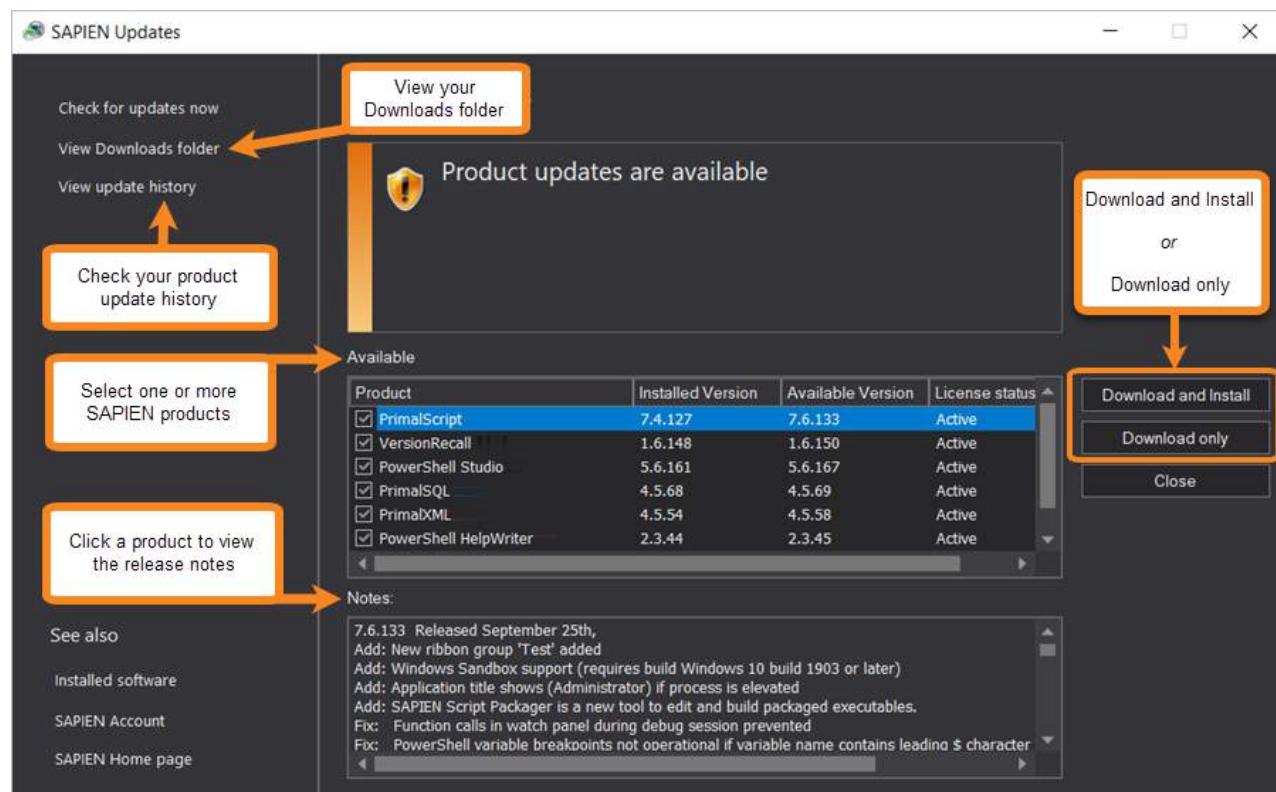
### How to access the SAPIEN Updates tool

- On the Help or Tools ribbon > click Check Now or Check For Updates (Updates section).

-OR-

- Click the [notification icon](#)  above the ribbon > then in the Notifications dialog, click the update notification.

### SAPIEN Updates Tool



SAPIEN Updates Tool

## SAPIEN Updates - Options

<b>Check for updates now</b>	Immediately checks to see if additional product updates are available.
<b>View Downloads folder</b>	Displays the Downloads folder in File Explorer.
<b>View update history</b>	Displays the history of all downloaded and installed product updates.
<b>Available</b>	Displays a selectable list of available product updates.  Select one or more products to Download or Download and Install.
<b>Download and Install</b>	Downloads and installs the updates for the product(s) selected in the <b>Available updates</b> list.
<b>Download only</b>	Downloads the updates for the product(s) selected in the <b>Available updates</b> list.
<b>Close</b>	Closes the SAPIEN Updates tool.
<b>Notes</b>	Displays a brief synopsis of what was changed, added, or fixed for the products selected in the <b>Available</b> window.  The build history for all SAPIEN products is <a href="#">available here</a> .

## Update On-Demand

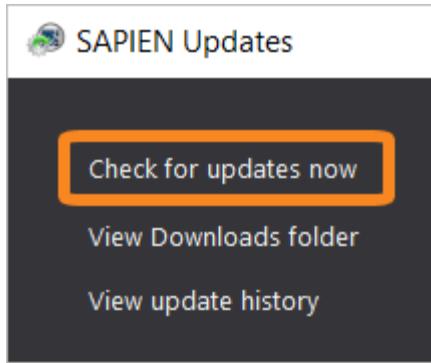
You don't need to wait to be notified when an update is available; you can check for updates at any time. This is particularly useful if you've heard about a new update and want to install it immediately, or if you are ready to start a new project and want to complete all updates before you begin.

### How to check for updates on-demand

- On the Help or Tools ribbon > select **Check Now** or **Check For Updates** to open the SAPIEN Updates tool.

 These instructions may vary between SAPIEN products.

- In the SAPIEN Updates tool, select **Check for updates now**:



The latest product updates are displayed in the SAPIEN Updates Available window.

## Security and Permissions

---

Installing updates to programs in a Program Files directory requires the permissions of a member of the Administrators group on the computer. When you click **Download and Install** in the SAPIEN Updates tool, or if you install after downloading, you will be prompted for administrator credentials.

The update tool requires a functioning internet connection and unimpeded access through your internet firewall. For some installations, you might need to create a firewall rule to allow access or make some accommodations.

## 16.2 Keyboard Shortcuts

This section covers the keyboard shortcuts available in PowerShell Studio.

## General Commands

---

Copy	<i>Ctrl + C</i>
Paste	<i>Ctrl + V</i>
Cut	<i>Ctrl + X</i>
Select All	<i>Ctrl + A</i>
Delete	<i>Del</i>
Undo	<i>Ctrl + Z</i>
Redo	<i>Ctrl + Y</i>
New File	<i>Ctrl + N</i>
New Project	<i>Ctrl + Shift + N</i>
Open File	<i>Ctrl + O</i>
Open Project	<i>Ctrl + Shift + O</i>
Save	<i>Ctrl + S</i>
Save All	<i>Ctrl + Shift + S</i>
Print	<i>Ctrl + P</i>
Help	<i>F1</i>
Switch to Next Document Tab	<i>Ctrl + Tab</i>
Switch to Prev Document Tab	<i>Ctrl + Shift + Tab</i>
Minimize Ribbon	<i>Ctrl + F1</i>
Access Ribbon Key Shortcuts	<i>Alt</i>
Find in Files	<i>Ctrl + Shift + F</i>

## Document Commands

---

Build All	<i>F7</i>
Build Package	<i>Ctrl + F7</i>
Build MSI	<i>Shift + F7</i>
Deploy Files	<i>Ctrl + Shift + F7</i>
Preview Form	<i>Ctrl + Shift + F5</i>
Debug	<i>F5</i>
Run	<i>Ctrl + F5</i>
Run Remote	<i>F6</i>
Run Remote RSEE	<i>Shift + F6</i>
Stop Script	<i>Shift + F5</i>
Run in Console	<i>Ctrl + F8</i>
Run Selection	<i>Shift + F8</i>
Run Selection in Console	<i>F8</i>
Format Script	<i>Ctrl + Shift + J</i>

## Navigation Commands

---

Navigate Backward	<i>Ctrl + Shift + Minus</i>
Navigate Forward	<i>Ctrl + Shift + Plus</i>

## Debugging Commands

---

Debug Document	<i>F5</i>
Debug with Multiple Files	<i>Ctrl + M</i>
Debug Remote (RSEE)	<i>Ctrl + F6</i>
Resume	<i>F5</i>
Step Into	<i>F11</i>
Step Over	<i>F10</i>
Step Out	<i>Shift + F11</i>
Run to Cursor	<i>Ctrl + F10</i>
Toggle Breakpoint	<i>F9</i>
Delete all Breakpoints	<i>Ctrl + Shift + F9</i>
Toggle Breakpoint Enable / Disable	<i>Shift + F9</i>
Toggle Tracepoint	<i>Ctrl + F9</i>
Delete all Tracepoints	<i>Ctrl + Shift + Alt + F9</i>

## Designer Commands

---

Switch between Design and Editor	<i>Ctrl + D</i>
Move Up	<i>Up Arrow</i>
Move Left	<i>Left Arrow</i>
Move Right	<i>Right Arrow</i>
Move Down	<i>Down Arrow</i>
Nudge Up	<i>Ctrl + Up Arrow</i>
Nudge Left	<i>Ctrl + Left Arrow</i>
Nudge Right	<i>Ctrl + Right Arrow</i>
Nudge Down	<i>Ctrl + Down Arrow</i>
Height Increase	<i>Shift + Up Arrow</i>
Height Decrease	<i>Shift + Down Arrow</i>
Width Increase	<i>Shift + Right Arrow</i>
Width Decrease	<i>Shift + Left Arrow</i>
Nudge Height Increase	<i>Ctrl + Shift + Up Arrow</i>
Nudge Height Decrease	<i>Ctrl + Shift + Down Arrow</i>
Nudge Width Increase	<i>Ctrl + Shift + Right Arrow</i>
Nudge Width Decrease	<i>Ctrl + Shift + Left Arrow</i>
Bring to Front	<i>Ctrl + B</i>
Send to Back	<i>Ctrl + Shift + B</i>
Cancel	<i>Escape</i>
Reverse Cancel	<i>Shift + Escape</i>
Add Default Action Event	<i>Enter</i>
Add Event	<i>Ctrl + E</i>
Apply Style	<i>Ctrl + L</i>
Create Style	<i>Ctrl + Shift + L</i>
Create Control Set	<i>Ctrl + T</i>

## Editor Commands

---

Go To Line	<i>Ctrl + G</i>
Find / Replace	<i>Ctrl + F</i>
Find / Replace	<i>Ctrl + H</i>
Find Next	<i>F3</i>
Find Previous	<i>Shift + F3</i>
Find Selection Next	<i>Ctrl + F3</i>
Find Selection Previous	<i>Ctrl + Shift + F3</i>
Find All References	<i>Ctrl + Alt + F</i>
Comment Line	<i>Ctrl + Q</i>
Comment Multiline	<i>Ctrl + Shift + Alt + Q</i>
Un-Comment Line	<i>Ctrl + Shift + Q</i>
Go To Next Bookmark	<i>F2</i>
Go To Previous Bookmark	<i>Shift + F2</i>
Toggle Bookmark	<i>Ctrl + F2</i>
Clear All Bookmarks	<i>Shift + Ctrl + F2</i>
Toggle Collapsed Code	<i>Ctrl + Shift + M</i>
Collapse All Code Nodes	<i>Ctrl + Minus</i>
Expand All Code Nodes	<i>Ctrl + Plus</i>
Create Region	<i>Ctrl + R</i>
Copy as HTML	<i>Ctrl + Shift + C</i>
Wrap the selected text in () (parentheses)	<i>Ctrl + Shift + 9</i>
Wrap the selected text in [] (square brackets)	<i>Ctrl + [</i>
Wrap the selected text in {} (curly braces)	<i>Ctrl + Shift + [</i>
Wrap the selected text in '' (single quotes)	<i>Ctrl + '</i>
Wrap the selected text in "" (double quotes)	<i>Ctrl + Shift + '</i>
Wrap the selected text in \$() (sub-expression)	<i>Ctrl + Shift + Alt + 9</i>
Toggle string quotes	<i>Ctrl + Alt + '</i>

## *Return Commands*

---

Insert Line Break	<i>Enter</i>
Insert Line Break	<i>Shift + Enter</i>
Open Line Above	<i>Ctrl + Enter</i>
Open Line Below	<i>Ctrl + Shift + Enter</i>

## *Delete / Backspace Commands*

---

Delete	<i>Del</i>
Delete Line	<i>Ctrl + Shift + L</i>
Delete To Next Word	<i>Ctrl + Del</i>
Backspace	<i>Backspace</i>
Backspace	<i>Shift + Backspace</i>
Backspace To Previous Word	<i>Ctrl + Backspace</i>

## *Clipboard / Undo Commands*

---

Copy To Clipboard	<i>Ctrl + C</i>
Copy To Clipboard	<i>Ctrl + Ins</i>
Cut Line To Clipboard	<i>Ctrl + L</i>
Cut To Clipboard	<i>Ctrl + X</i>
Cut To Clipboard	<i>Shift + Del</i>
Paste From Clipboard	<i>Ctrl + V</i>
Paste From Clipboard	<i>Shift + Ins</i>
Undo	<i>Ctrl + Z</i>
Redo	<i>Ctrl + Y</i>
Redo	<i>Ctrl + Shift + Z</i>

## *Movement Commands*

---

Move Down	<i>Down</i>
Move Up	<i>Up</i>

Move Left	<i>Left</i>
Move Right	<i>Right</i>
Move To Previous Word	<i>Ctrl + Left</i>
Move To Next Word	<i>Ctrl + Right</i>
Move To Line Start	<i>Home</i>
Move To Line End	<i>End</i>
Move To Document Start	<i>Ctrl + Home</i>
Move To Document End	<i>Ctrl + End</i>
Move Page Up	<i>Page Up</i>
Move Page Down	<i>Page Down</i>
Move To Visible Top	<i>Ctrl + Page Up</i>
Move To Visible Bottom	<i>Ctrl + Page Down</i>
Move To Matching Bracket	<i>Ctrl + ]</i>
Move To Next Modified Line	<i>Ctrl + Shift + Down</i>
Move To Previous Modified Line	<i>Ctrl + Shift + Up</i>
Go To Next Occurance	<i>Ctrl + Shift + Alt + Down</i>
Go To Previous Occurance	<i>Ctrl + Shift + Alt + Up</i>
Go To Last Edit Position	<i>Ctrl + E</i>
Go To Function Declaration	<i>F12</i>
Go To Next Function	<i>Shift + F12</i>
Go To Prev Function	<i>Ctrl + Shift + F12</i>

## *Scroll Commands*

---

Scroll Down	<i>Ctrl + Down</i>
Scroll Up	<i>Ctrl + Up</i>

## *Indenting Commands*

---

Indent	<i>Tab</i>
Indent	<i>Alt + Right</i>
Outdent	<i>Shift + Tab</i>

Outdent

*Alt + Left*

## ***Selection Commands***

---

Select Down	<i>Shift + Down</i>
Select Up	<i>Shift + Up</i>
Select Left	<i>Shift + Left</i>
Select Right	<i>Shift + Right</i>
Select To Previous Word	<i>Ctrl + Shift + Left</i>
Select To Next Word	<i>Ctrl + Shift + Right</i>
Select To Line Start	<i>Shift + Home</i>
Select To Line End	<i>Shift + End</i>
Select To Document Start	<i>Ctrl + Shift + Home</i>
Select To Document End	<i>Ctrl + Shift + End</i>
Select Page Up	<i>Shift + Page Up</i>
Select Page Down	<i>Shift + Page Down</i>
Select To Visible Top	<i>Ctrl + Shift + Page Up</i>
Select To Visible Bottom	<i>Ctrl + Shift + Page Down</i>
Select All	<i>Ctrl + A</i>
Select Word	<i>Ctrl + W</i>
Cut Word	<i>Ctrl + Shift + W</i>
Select To Matching Bracket	<i>Ctrl + Shift + ]</i>
Select Block Down	<i>Shift + Alt + Down</i>
Select Block Up	<i>Shift + Alt + Up</i>
Select Block Left	<i>Shift + Alt + Left</i>
Select Block Right	<i>Shift + Alt + Right</i>
Select Block To Previous Word	<i>Ctrl + Shift + Alt + Left</i>
Select Block To Next Word	<i>Ctrl + Shift + Alt + Right</i>

## ***PrimalSense™ Commands***

---

PrimalSense™ Complete Word

*Ctrl + Space*

PrimalSense™ Show Method Parameter Info	<i>Ctrl + Shift + Space</i>
Trigger Custom PrimalSense™ Keyword	<i>Ctrl + Alt + P</i>

### ***Other Commands***

---

Change Character Casing (to uppercase)	<i>Ctrl + Shift + U</i>
Change Character Casing (to lowercase)	<i>Ctrl + U</i>
Collapse Selection	<i>Escape</i>
Toggle Overwrite Mode	<i>Insert</i>
Transpose Characters	<i>Ctrl + T</i>
Transpose Words	<i>Ctrl + Shift + T</i>
Transpose Lines	<i>Ctrl + Shift + Alt + T</i>
Expand All Alias to Cmdlets	<i>Ctrl + Shift + A</i>
Toggle Alias	<i>Ctrl + B</i>
Insert Snippet	<i>Ctrl + K</i>
Surround With Snippet	<i>Ctrl + Shift + K</i>
Expand Snippet Shortcut	<i>Ctrl + J</i>
Insert New Function	<i>Ctrl + Shift + E</i>
Edit Function	<i>Ctrl + Shift + Alt + E</i>
Rename Object	<i>Ctrl + Alt + J</i>
Edit Script Parameters	<i>Ctrl + Shift + P</i>
Qualify cmdlet Names	<i>Ctrl + Shift + H</i>
Unqualify cmdlet Names	<i>Ctrl + Alt + H</i>
Splat Command	<i>Ctrl + Alt + S</i>

## Go-To Panel Commands

---

Call Stack	<i>Ctrl + Alt + P, K</i>
Console	<i>Ctrl + Alt + P, C</i>
Debug Console	<i>Ctrl + Alt + P, D</i>
Find Results	<i>Ctrl + Alt + P, R</i>
Function Explorer	<i>Ctrl + Alt + P, F</i>
Help	<i>Ctrl + Alt + P, H</i>
Object Browser	<i>Ctrl + Alt + P, B</i>
Output	<i>Ctrl + Alt + P, O</i>
Performance	<i>Ctrl + Alt + P, M</i>
Project	<i>Ctrl + Alt + P, J</i>
Properties	<i>Ctrl + Alt + P, P</i>
Snippets	<i>Ctrl + Alt + P, S</i>
Toolbox	<i>Ctrl + Alt + P, T</i>
Tools Output	<i>Ctrl + Alt + P, L</i>
Variables	<i>Ctrl + Alt + P, V</i>
Watch	<i>Ctrl + Alt + P, W</i>
Editor / Document	<i>Ctrl + Alt + P, E</i>

## 16.3 Appendices

### Appendices for PowerShell Studio Help Manual

---

[Appendix A: Manual and Product Version](#) 

[Appendix B: Icon License Attribution](#) 

### 16.3.1 Appendix A: Manual Version

## Appendix A Manual Version

---

This help manual is in the process of being updated. Some features and images in this manual version may not reflect the current product functionality.

#### **Blog articles**

For the latest product tips and feature demonstrations, check out the PowerShell Studio articles on the [SAPIEN blog](#).

#### **Release details**

To view a brief description of what was changed, added, or fixed in the most recent PowerShell Studio builds, view the product [version history](#).

#### **Need more help?**

Please direct your product related questions to the [PowerShell Studio support forum](#), and your scripting questions to the appropriate [Scripting Answers forum](#).

---

### 16.3.2 Appendix B: Icon License Attribution

## Appendix B Icon License Attribution

---

Icons used in this manual are licensed under [Creative Commons Attribution 3.0 Unported \(CC BY 3.0\)](#).

Icons made by [Icomoon](#) from [www.flaticon.com](#) are licensed under [CC 3.0 BY](#):

 [Thumb up](#)

 [Information](#)

 [Warning](#)

