PRIME
Parameters, Expressions, and Conditions User Guide

Version 5.1

March 2025

SINCE DAY ONE.

Chyron PRIME Parameters, Expressions, and Conditions User Guide • 5.1 • March 2025
This document is distributed by Chyron in online (electronic) form only, and is not available for purchase in printed form.

Table of Contents

Parameters/Expressions

Introduction

PRIME supports both parameters and expressions.

At its most basic level, a **parameter** is a container for a specific type of data that may be leveraged by other elements of a scene. Parameters reference values directly; e.g. an Integer parameter that is set to the number 10. The value of this parameter will never change from 10 unless the parameter itself is changed.

On the contrary, an **expression** may be thought of as a formula that is evaluated dynamically; e.g. an Integer expression may be set to the X position of an object in the scene. As a result, the value of the expression will automatically change as the object is moved along the X-axis. Scenes have their own distinct collection of parameters and expressions that are transient in nature. This means that initial values that are set while working with the designer are lost once the scene has been cleared. Each project maintains a collection of parameters, and updates to these parameters, that persist in real time. These parameters are global to all scenes within the project.

Parameter Scopes

Application Parameters

The scope of Application Parameters is global meaning this parameter type will be accessible to every Project and every scene within every project.

Project Parameters

Project parameters are only accessible to every scene in the current project the parameter is created in.

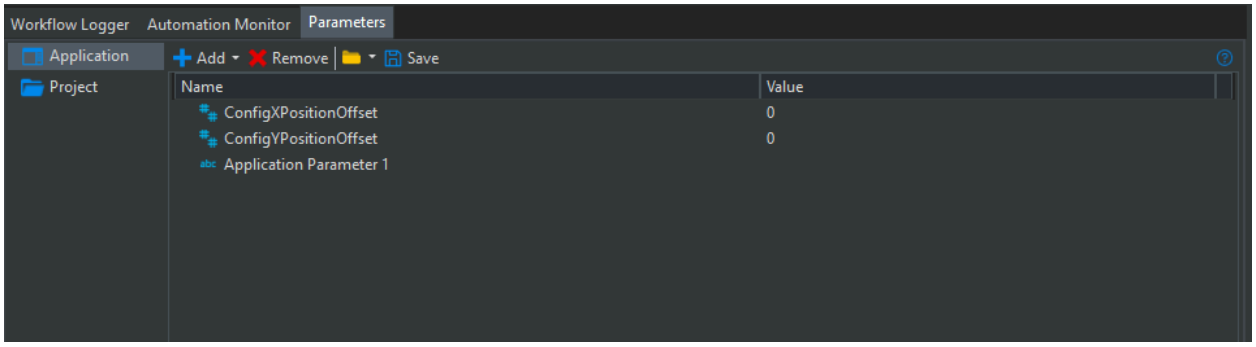
Scene Parameters

Scene parameters are only accessible to the individual scene they are created in.

PRIME Playout

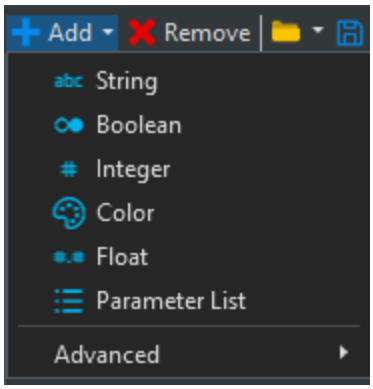
In Prime Playout you are able to view Application and Project Parameters.

To view Application parameters as well as the Project parameters of the currently active project select **View > Parameters**.



Add New Parameter

Clicking the **Add** button will create a new parameter at the scope you have selected, either Application or Project. The default type of the new parameter is String (text) with a null default value.



Click the down arrow, to the right of the **Add** ↓ button to select a different parameter type to add to the scope selected, either Application or Project.

See Parameter Type Summary for more information

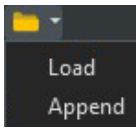
Remove Parameter

Clicking the **Remove** button will delete the currently selected parameter from the scope you have selected (either project or application).

Save Parameters

Click the **Save** button to save all the parameters in the selected scope (either project or application) as an XML file.

Import Parameter XML file



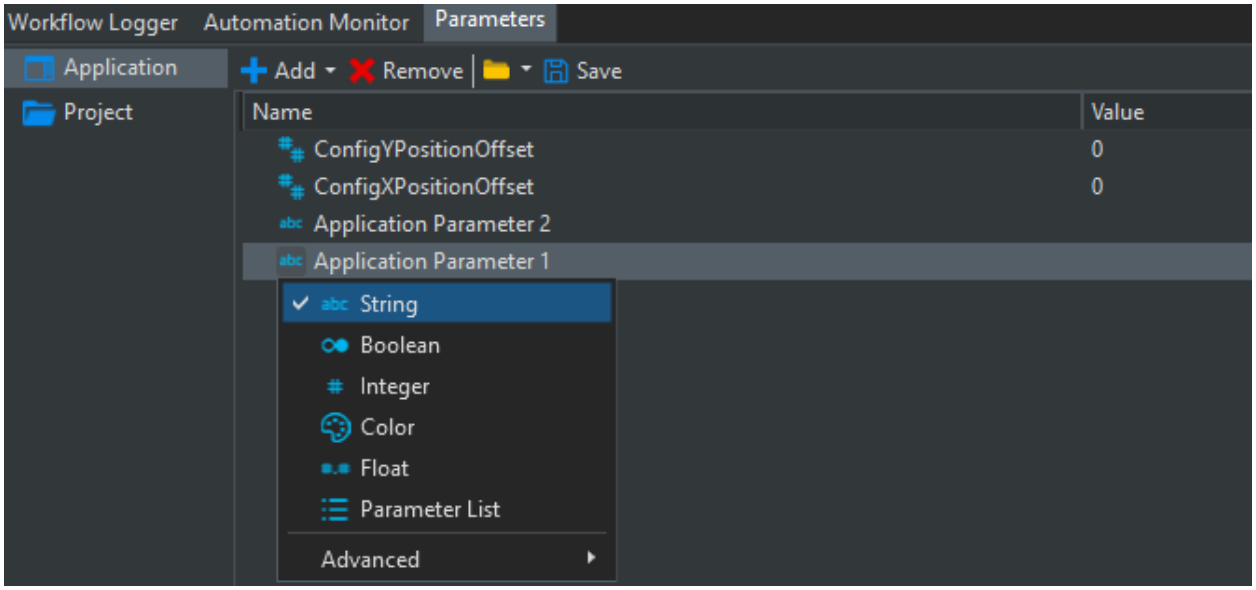
Click the **Import** button to Load or Append a Parameter XML file.

Select **Load** to only display parameters saved in the XML file.

Select **Append** to add parameters in the XML file to the current display of parameters.

Edit Parameter

Click Parameter type icon to change the type of a parameter

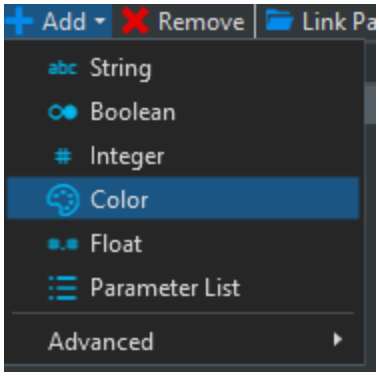


To edit the name or value of a Parameter click into the cell you wish to update.

PRIME Editor

Adding New Parameter

Clicking the **Add** button will create a new scene parameter. The default type the new parameter is String (text) with a null default value.



Click the down arrow, to the right of the **Add** ↓ button to select a different parameter type to add to the scope selected, either Application or Project.

See Parameter Type Summary for more information

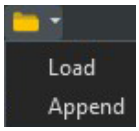
Remove Parameter

Clicking the **Remove** button will delete the currently selected parameter.

Link Parameter

Clicking the **Link Parameter** button lets you navigate to an existing Project or Application parameter that can then be linked to the active scene.

Import Parameter XML file



Click the **Import** button to Load or Append a Parameter XML file.

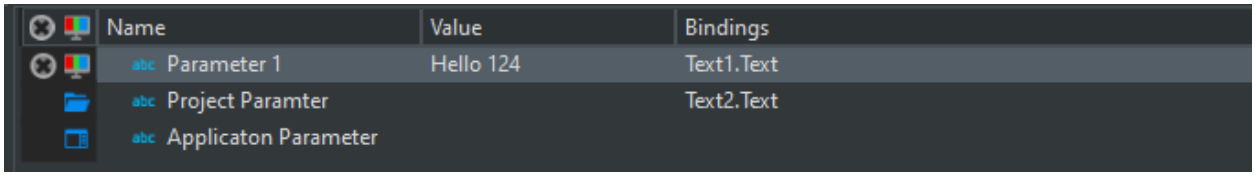
Select **Load** to only display parameters saved in the XML file.

Select **Append** to add parameters in the XML file to the current display of parameters.

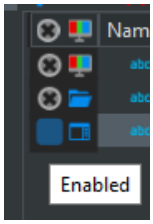
Save Parameters

Click the **Save** button to save parameters as an XML file.

Edit Parameters

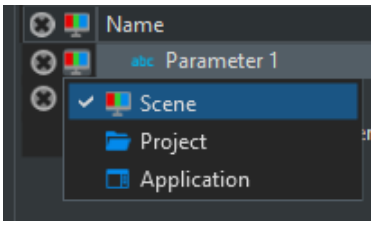


Disable / Enable Parameter



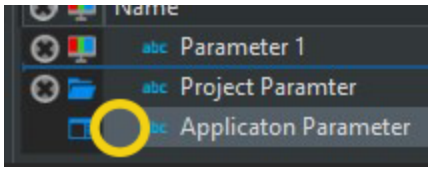
By default any new parameter will be enabled. Press the negative space in the enable column to disable the selected parameter.

Modify Parameter Scope



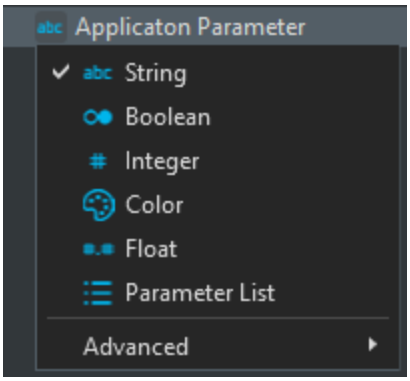
Click the scope icon of the parameter you wish to modify. This will reveal a list of available scopes to select from; Scene, Project and Application.

Modify Parameter Order



Hold down mouse click left, on the negative space to the left of the Parameter name. Drag and drop parameter to desired position in the list.

Parameter Type



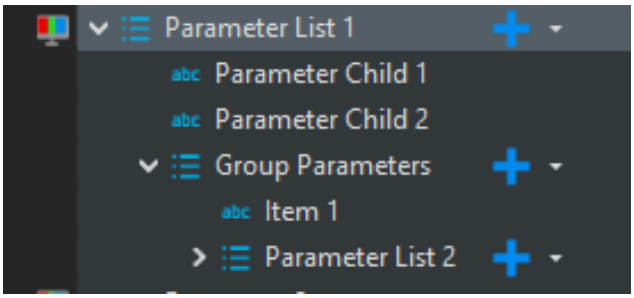
Click the parameter type icon of the parameter you wish to modify. This will reveal a list of available parameter types. Modifying the type of an existing parameter with a set value will attempt to convert the current value to one appropriate for the new type. If the data cannot be converted, then the new value will be defaulted accordingly.

Parameter Type Summary

Type	
String	Sequence of Alphanumeric characters
Boolean	Can only have one of two values, True or False
Integer	Whole Numbers without decimals
Color	Color [Black]
Float	Stores Fractional Numbers with one or more decimal
ByteArray	Integers in the range between 0 and 255
DateTime	1/1/0001 12:00:00 AM
Double	Fractional numbers. Sufficient for storing 15 decimal digits
Long	Whole numbers from -9223372036854775808 to 9223372036854775807. Used when int is not large enough
TimeSpan	Time interval and can be expressed as a particular number of days, hours, minutes, seconds, and milliseconds

Parameter List

A Parameter list is a container, where parameters within the same scope can be “grouped”. This allows you to organize parameters together, rather than having a single long list of parameters.



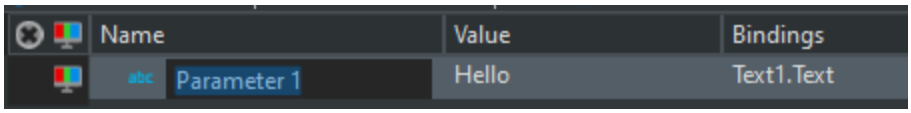
To add a new parameter into a Parameter list press the + icon to the right of the parameter list. Existing parameters can be dragged and dropped into a Parameter List.

You can only add parameters into a parameter list that have the same scope.

Parameter Lists can be nested by drag and dropping a Parameter list into another Parameter list.

Parameter Name

Click in the parameter name field to edit the parameter name. Each parameter must have a unique name. However, uniqueness is only guaranteed within the context of the project parameter collection. Scenes may include parameters with names that match project parameters because these are addressed differently.



Parameter Value

Type	Default
String	Empty string
Boolean	False
Integer	0
Color	Color [Black]

Float	0
ByteArray	Empty byte array
DateTime	1/1/0001 12:00:00 AM
Double	0.0
Long	0
TimeSpan	TimeSpan with 0 Ticks

Sample Value Conversion

Whenever possible, existing parameter values will be converted.

Initial Type	Current Value	Modified Type	New Value
Integer	1	Double	1.0
Integer	1	Boolean	True
Integer	0	Boolean	False
String	1	Double	1.0
String	Hello	Double	0
Double	1.7	Integer	1

Bindings

Parameter values may be bound directly to one or more object properties within a scene. These bindings appear in the **Bindings** column and may be modified either by direct text entry into the bindings field, or through a drag-and-drop operation. Scene parameters may only be bound to objects within the same scene. However, project parameters may be bound to objects of any scene within the project. Project parameters may be viewed and modified in the designer by clicking the **Project** text within the parameters panel.

Bindings allow a parameter to automatically update one or more targets whenever the parameter value is changed. For example, the user may set up an Integer parameter that is bound to the opacity of an object within the scene. When the parameter is updated, the opacity of the bound object will update as well.

Manual Bindings

The user may bind a parameter directly to an object by clicking the binding field within the parameter grid control and typing text that describes a valid target. Target text has several variations, but all conform to two standards: (1) A single period character separates fields and (2) names are trimmed to remove white space. For example:

Target Type	Scene Example
Object Property	Text1.Text
Keyframe Property	Text1.Action1.Keyframe1.PositionX

Notice that while an action may be named **Action 1,** the target text removes the space and uses **Action1 instead**. In general, a scene parameter can either be bound to a property with the form *ObjectName.PropertyName* or to a specific property of a keyframe using the form *ObjectName.ActionName.KeyframeName.PropertyName*. Because scene parameters are actual objects within the scene, they are also valid binding targets. This means that one parameter can be bound to the value of another (example target text: Parameter1.Value). Project parameters add one stipulation to bindings in that their bindings may only target scenes explicitly or other project parameters.

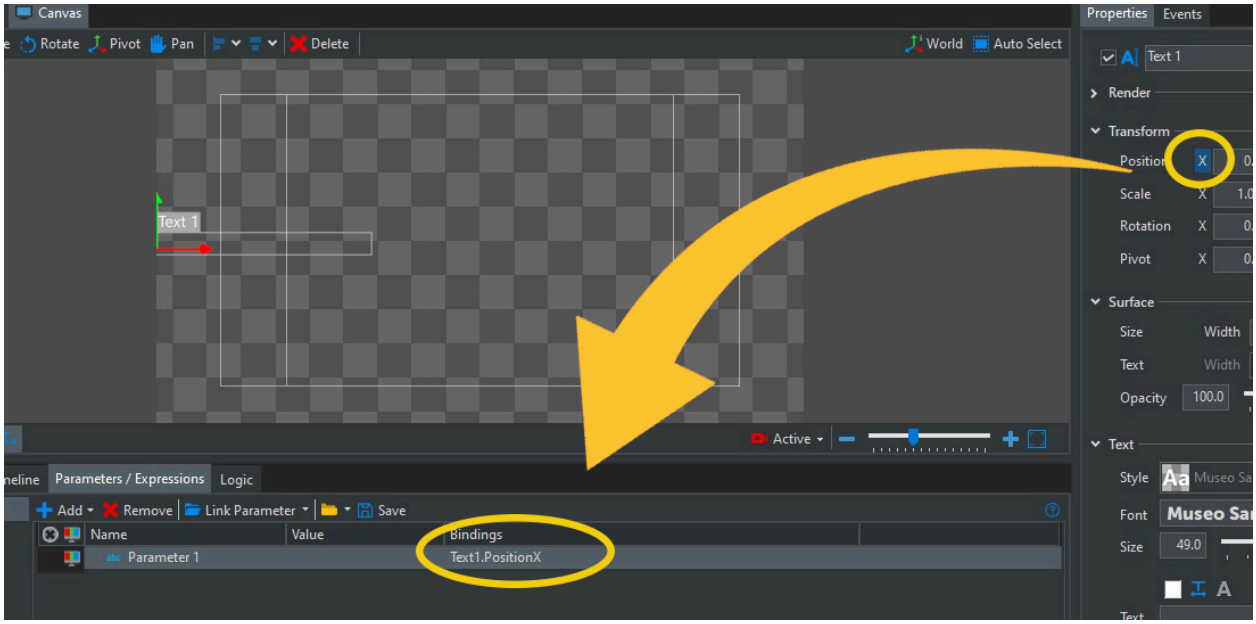
Target Type	Project Example
Object Property	SceneName.Text1.Text
Keyframe Property	SceneName.Text1.Action1.Keyframe1.PositionX
Project Parameter	Project.Parameter1

Drag-and-Drop Binding

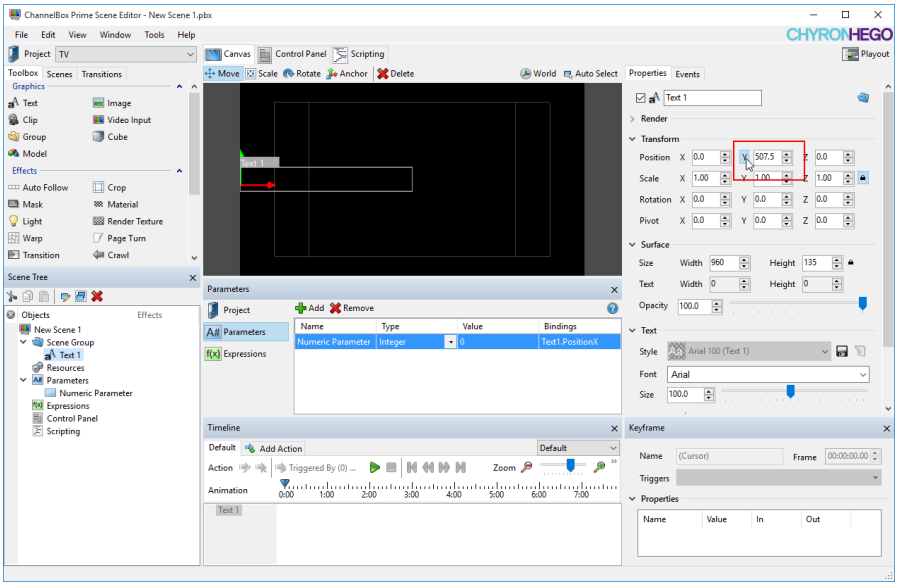
To facilitate binding object properties to parameters, the user may utilize drag-and-drop from three locations within the designer.

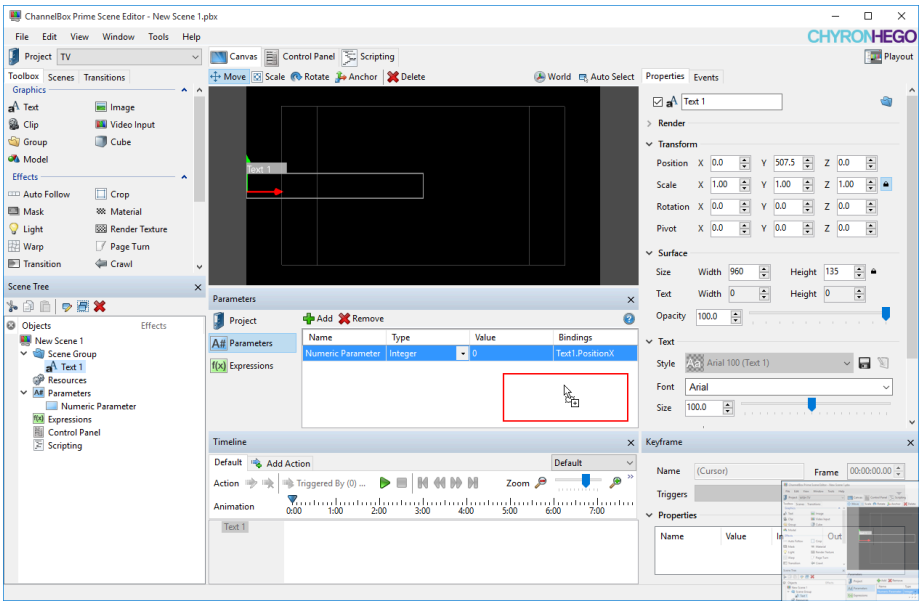
Option 1, Dragging from the Properties Panel

In the property panel, hover over the property you wish to bind the parameter to. Holding left click on the mouse drag and drop the individual item directly onto the Parameters panel.



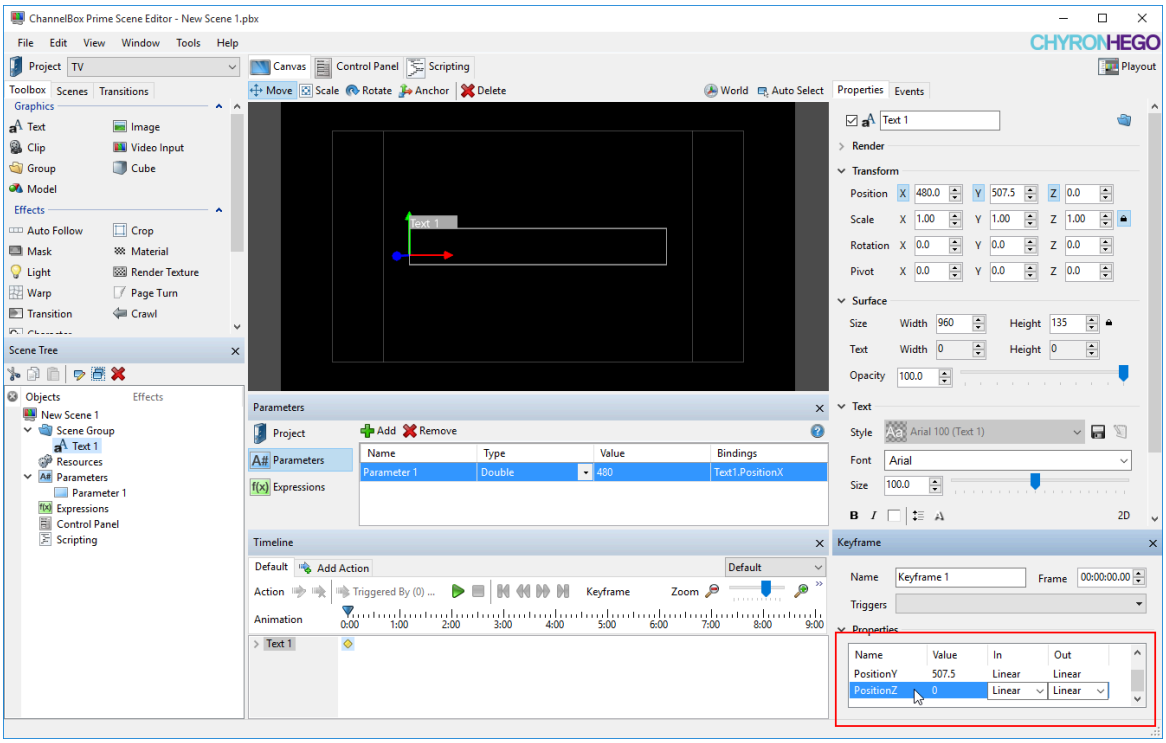
Alternatively, the property can drop onto the parameter panel itself. This will create a new parameter already bound to the property being dragged and with a parameter type that matches the type of the property.





Option 2, Dragging from the Keyframe Panel

Similarly, the user may drag properties directly from the Keyframe panel.



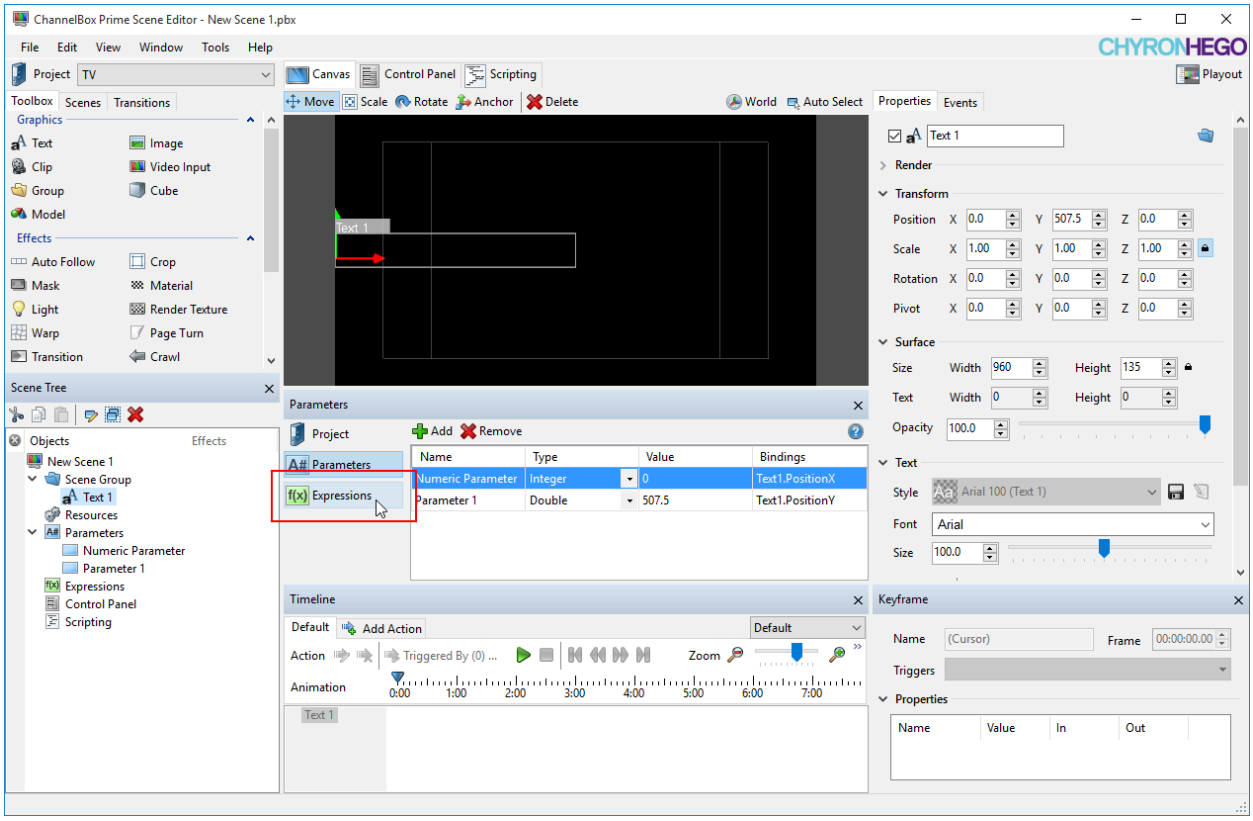
Option 3, Dragging from the Scene Tree

The user may also drag objects from the scene tree directly onto the Parameters panel. This will generate target text for bindings based on the default property of the object dragged. For example, dragging a text object will utilize the Text property (example target text: Text1.Text) while dragging a file picker control will utilize the File property (example target text: FilePicker1.File).

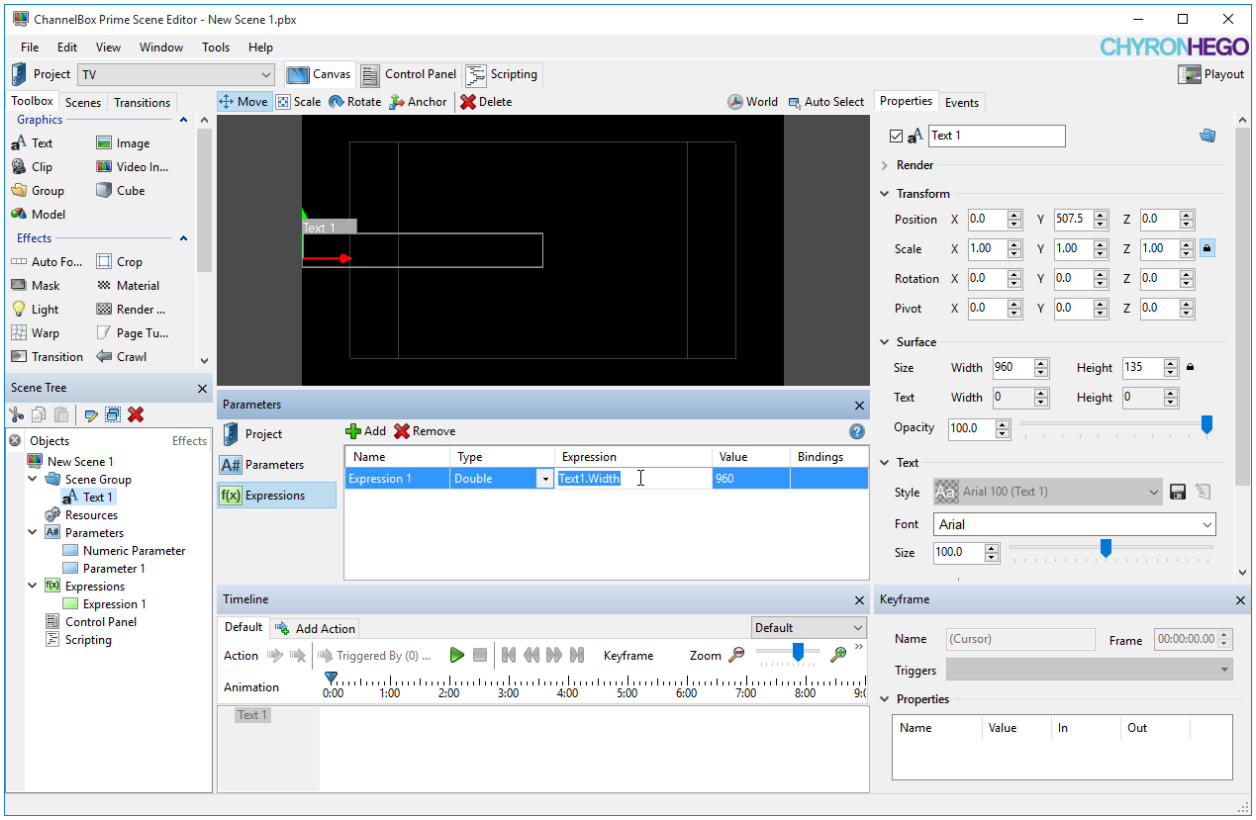
Expressions

Expressions are specialized parameters that evaluate dynamically, rather than having an explicit set value. Instead, the value is calculated based on a formula the user defines. Once configured, expressions support bindings in the same manner as parameters.

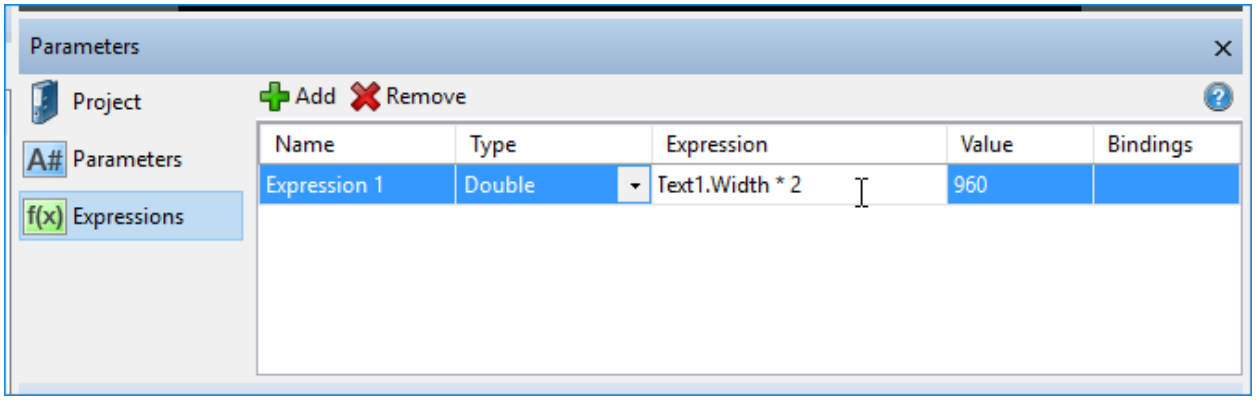
To view the expressions within a scene, click the **Expressions** text in the Parameters panel or click the Expressions node in the scene tree.



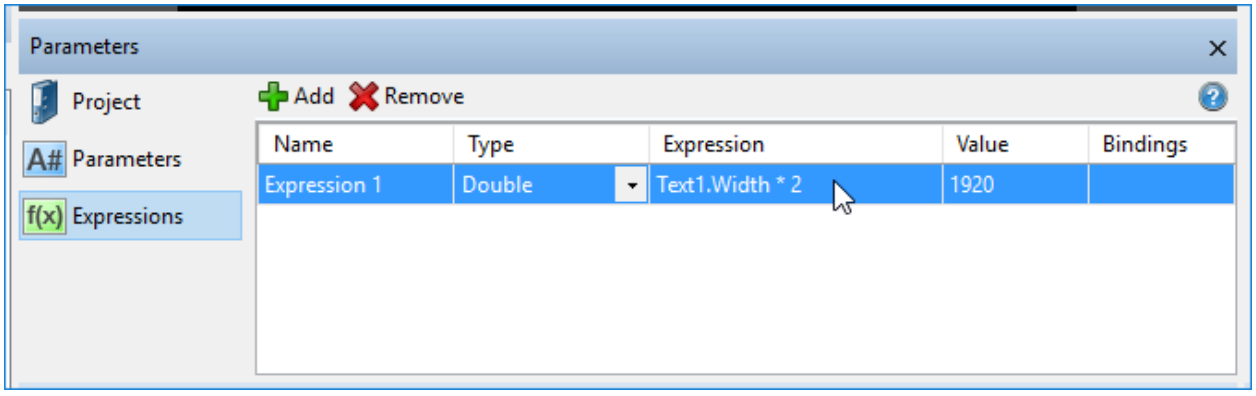
Expressions may be added, modified or removed in the same manner as scene or project parameters. However, the Value field of each expression is read only. Instead, a new field entitled Expression is available. The text in this field defines the formula that will be evaluated to determine the current value of the expression.



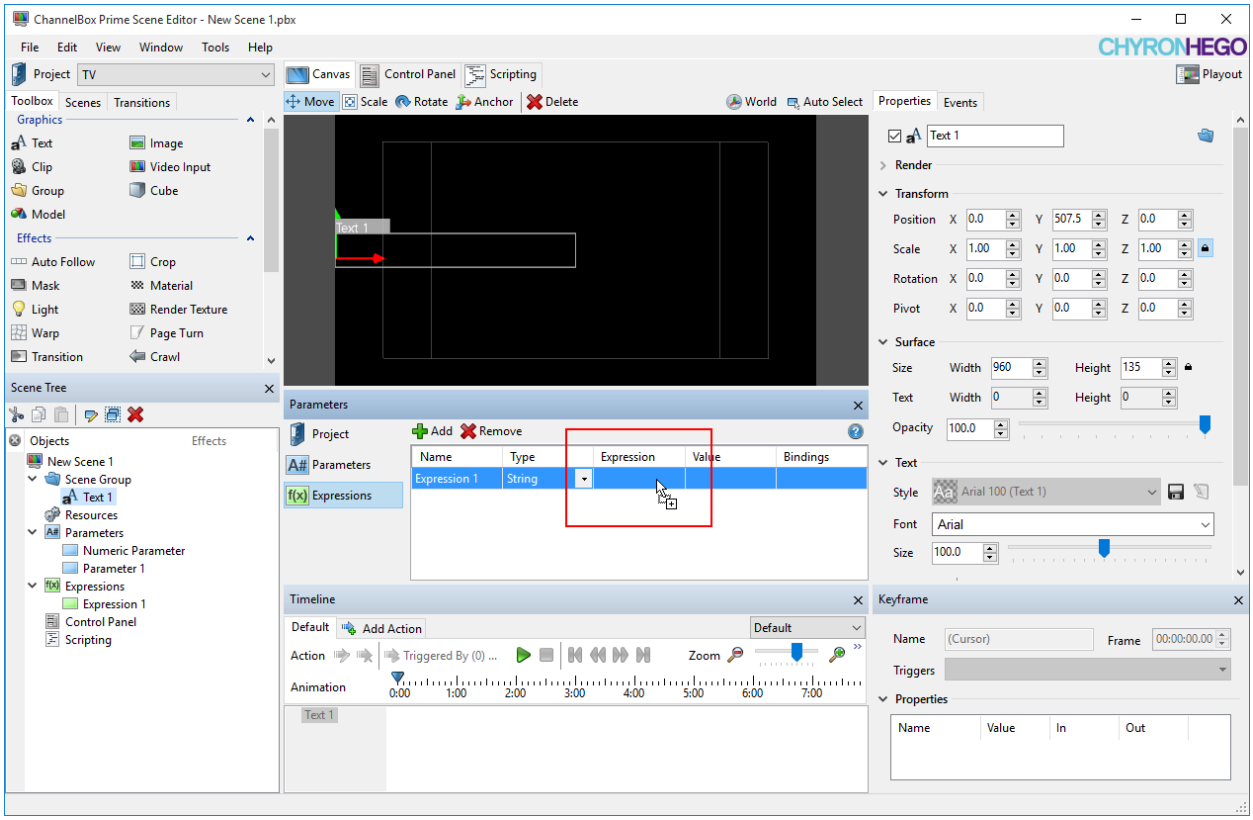
As with bindings, the expression text utilizes a naming convention for incorporating object and keyframe properties. As in the above screenshot, the user may set the expression text to an object property (Text1.Width) and once the edit has been committed, the Value column will update immediately to reflect the new value (960).



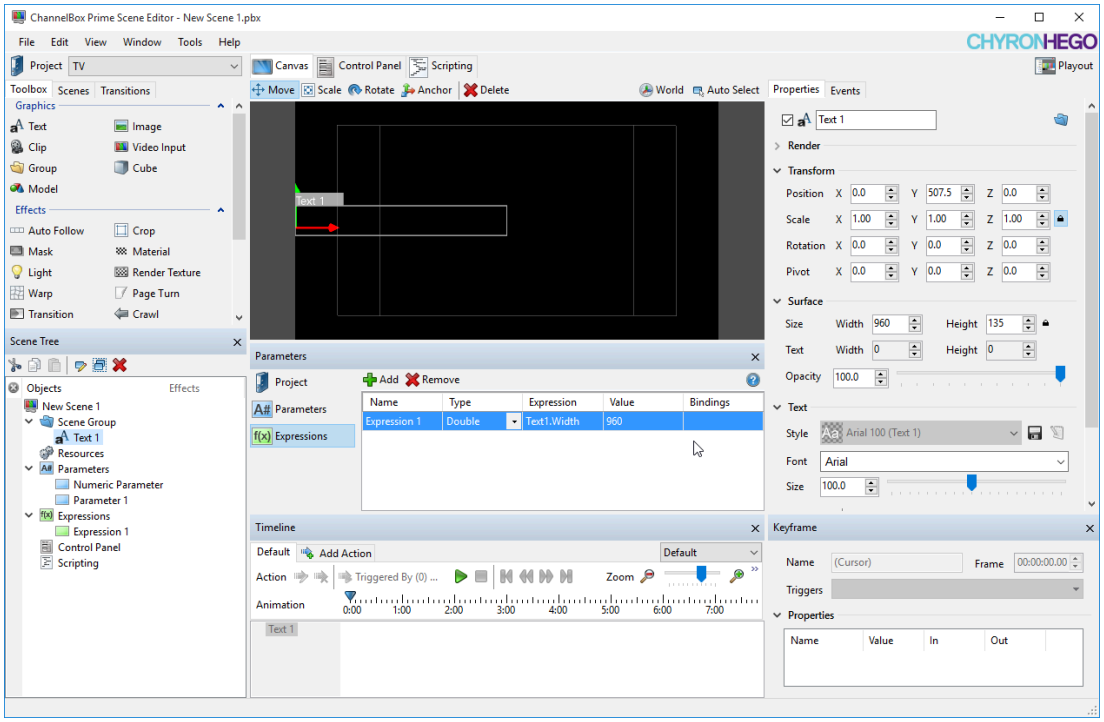
Unlike simple bindings, expression text can include a variety of mathematical, logical and string operations. Each expression can include references to one or more constant values (1, True, “ABC”) or variables (Text1.Width). For example, in the screenshot above the expression is being modified to multiply the width of *Text 1* by the number 2.



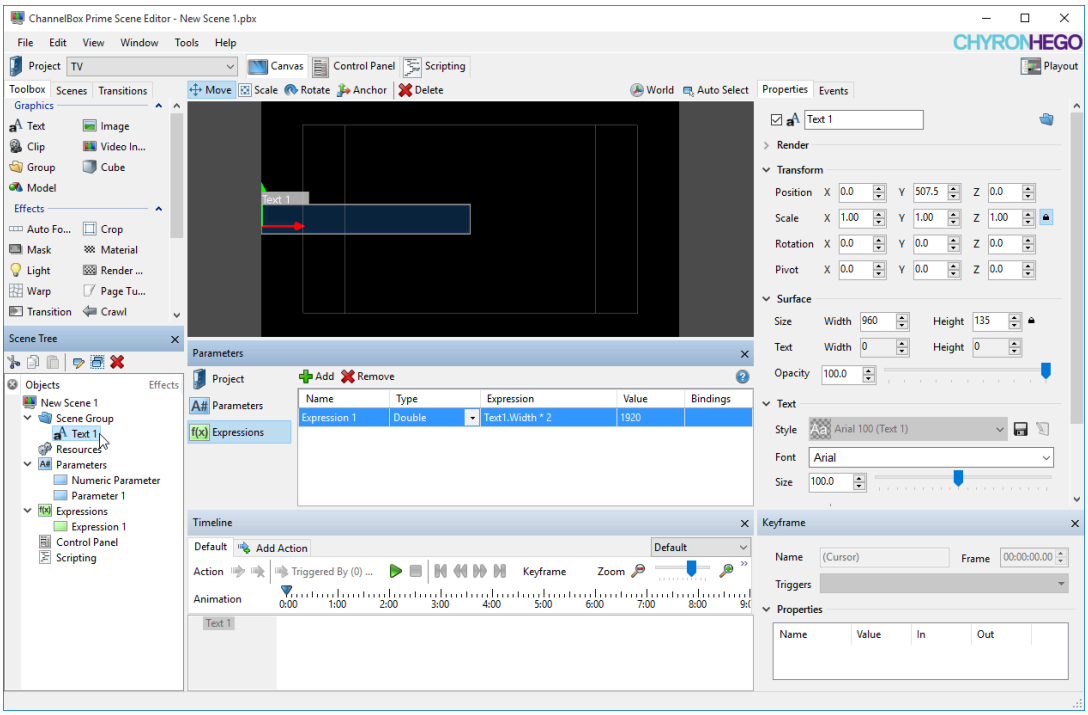
This evaluates to 1920 as indicated by the value column. As with bindings, the user may drag-and-drop directly onto the expression column of an existing expression.

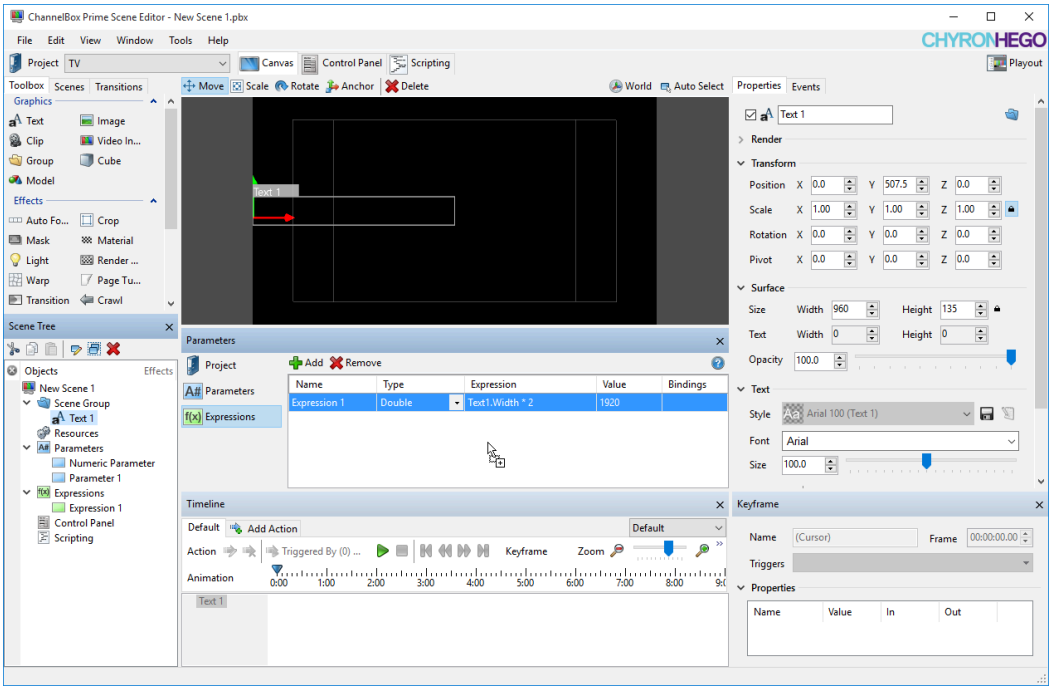


If the expression text is currently empty, the property will be added by itself. Otherwise, the property will be appended with an addition operator.

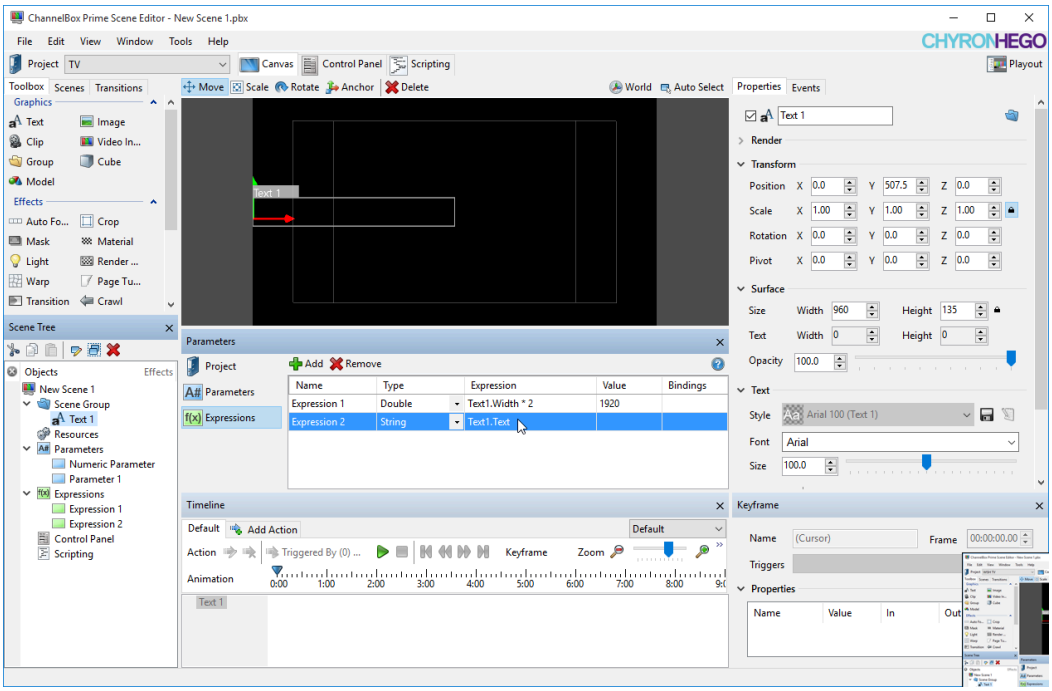


The user may also drag directly onto the grid control to create a new expression altogether.





The expression text will be set to the property being dragged and the type will match accordingly.



Expression Text

Expression text is evaluated left to right following an order of operations that is dependent upon the type of operation occurring within the expression. This simply means that evaluation is left to right except in the case of parentheses and mathematical operators.

Expression Text	Evaluation
7	Evaluates to the number 7.
Text1.Width	Evaluates to the current width of Text 1.
7 * Text1.Width	Evaluates to the number 7 multiplied by the current width of Text 1
7 * (Text1.Width + 3)	Evaluates to the number 7 multiplied by the result of adding 3 to the current width of Text 1.
Text1.PositionX + Text1.Width	Evaluates to the sum of Text 1's X-position and width.

Expression Language Support

Expressions support a number of logical, comparison, arithmetic and text operators as well as many mathematical functions.

Logical Operators

Operator	Usage	Meaning
and, &&	X **and** Y X **&&** Y	True if both the left and right operand are true. False otherwise. *This may only be used to compare Boolean values.*
or, ||	X **or** Y X **||** Y	True if either the left or right operand is true. False otherwise. *This may only be used to compare Boolean values.*
xor, ^	X **xor** X X **^** Y	True if only one of the operands is true. False otherwise. *This may only be used to compare Boolean values.*
not, !	**not**(X) **!**X	True if the operand is false. False otherwise. *This may only be used to compare Boolean values.*

Comparison Operators

The following operators may be used to compare two values within the expression.

Operator	Usage	Meaning
<	X **<** Y	True if the left operand is less than the right operand. *This may only be used to compare numeric values.*
<=	X **<=** Y	True if the left operand is less than or equal to the right operand. *This may only be used to compare numeric values.*
=	X **=** Y	True if both operands are equal.
!=	X **!=** Y	True if both operands are not equal.
>	X **>** Y	True if the left operand is greater than the right operand. *This may only be used to compare numeric values.*
>=	X **>=** Y	True if the left operand is greater than or equal to the right operand. *This may only be used to compare numeric values.*

Arithmetic Operators

The following operators may be used to perform basic arithmetic on numeric data values.

Operator	Usage	Meaning
+	X + Y	Performs an addition operation. *This may only be used with numeric values.*
-	X - Y	Performs a subtraction operation. *This may only be used with numeric values.*
*	X * Y	Performs a multiplication operation. *This may only be used with numeric values.*
/	X / Y	Performs a division operation. *This may only be used with numeric values.*
Mod	X Mod Y	Performs a modulus operation. *This may only be used with numeric values.*
^	X ^ Y	Raises the left operand to the exponential power denoted by the right operand. *This may only be used with numeric values.*
- (Unary)	-X	Multiplies the given number by -1.

String Operators

Operator	Usage	Meaning
+	X + Y	Concatenates two text strings together. Literal text values must be enclosed within double quotation marks (e.g. “ABC” + “DEF”).

Mathematical Functions

Function	Usage	Meaning
Maximum	Math.Max(X, Y)	Returns X or Y depending upon which value is larger.
Minimum	Math.Min(X, Y)	Returns X or Y depending upon which value is smaller.
Sign	Math.Sign(X)	Returns a value indicating the sign of X.
Absolute	Math.Sign(X)	Returns the absolute value of X.
Arc Sine	Math.Asin(X)	Returns the angle whose sine is X.
Arc Cosine	Math.Acos(X)	Returns the angle whose cosine is X.
Arc Tangent	Math.Atan(X)	Returns the angle whose tangent is X.
Hyperbolic Cosine	Math.Cosh(X)	Returns the hyperbolic cosine of angle X.
Hyperbolic Sine	Math.Sinh(X)	Returns the hyperbolic sine of angle X.
Hyperbolic Tangent	Math.Tanh(X)	Returns the hyperbolic tangent of angle X.
Square Root	Math.Sqrt(X)	Returns the square root of X.
Cosine	Math.Cos(X)	Returns the cosine of angle X.
Sine	Math.Sin(X)	Returns the sine of angle X.
Tangent	Math.Tan(X)	Returns the tangent of angle X.

Text Functions

Function	Usage	Meaning
Contains	Text1.Text.Contains("ABC")	Returns true if the text value "ABC" exists anywhere in the text value of Text1.Text.
EndsWith	Text1.Text.EndsWith("ABC")	Returns true if the text value of Text1.Text ends with the text "ABC"
IsEmpty	Text1.Text.IsEmpty()	Returns true if the text value of Text1.Text is either null, empty or all whitespace.
Length	Length(Text1.Text)	Returns the number of characters in the text value of Text1.Text
ReadFile	ReadFile("C:\Sample.txt")	Opens a text file and returns all lines.
Replace	Text1.Text.Replace("a", "b")	Replaces all instances of the text value "a" with the text value "b" in Text1.Text and returns the resulting value
StartsWith	Text1.Text.StartsWith("ABC")	Returns true if the text value of Text1.Text starts with the text "ABC"
Substring	Text1.Text.Substring(0, 5)	Retrieves a substring from Text1.Text
ToLower	Text1.Text.ToLower()	Returns a copy of Text1.Text converted to lowercase.
ToUpper	Text1.Text.ToUpper()	Returns a copy of Text1.Text converted to uppercase.
Trim	Text.Trim()	Returns a copy of Text1.Text with all leading and trailing whitespace removed.

Specialized Keywords

While expressions may interact with objects within a scene or execute global functions, they may also reference objects accessible through keywords that are context specific.

Channel: Refers to the channel upon which the current scene is loaded. Evaluating items that reference the Channel keyword is only supported in a playout environment; e.g. scenes open in the designer do not have a parent channel.

Item	Usage	Meaning
Index	Channel.Index	Returns the numeric index of the channel in which the executing scene is loaded.
Name	Channel.Name	Returns the text name of the channel in which the executing scene is loaded.
CloseScene	Channel.CloseScene(SceneName) **SceneName**: Text	Attempts to close a scene with the specified name or file path. This will only affect the channel in which the executing scene is loaded.
IsDescriptionAndLayerOnOutput	Channel.IsDescriptionAndLayerOnOutput(Description, Layer) **Description**: Text **Layer**: Specific number or text like "1-3"	Returns a value indicating whether a scene with the specified description is loaded currently on the specified layer within the current channel of the executing scene. The matching scene must be different from the executing scene.
IsDescriptionOnOutput	Channel.IsDescriptionOnOutput(Description) **Description**: Text	Returns a value indicating whether a scene with the specified description is loaded currently within the current channel of the executing scene. The matching scene must be different from the executing scene.
IsLayerOnOutput	Channel.IsLayerOnOutput(Layer) **Layer**: Specific number or text like "1-3"	Returns a value indicating whether a scene is loaded currently on the specified layer within the current channel of the executing scene. The matching scene must be different from the executing scene.
IsSceneAndLayerOnOutput	Channel.IsSceneAndLayerOnOutput(Name, Layer) **Name**: Text **Layer**: Specific number or text like "1-3"	Returns a value indicating whether a scene with the specified name is loaded currently on the specified layer within the current channel of the executing scene. The matching scene must be different from the executing scene.

IsSceneOnOutput	Channel.IsSceneOutput(Name) **Name**: Text	Returns a value indicating whether a scene with the specified name is loaded currently within the current channel of the executing scene. The matching scene must be different from the executing scene.
LoadScene	Channel.LoadScene(SceneName) **SceneName**: Text	Attempts to load a scene with the specified name or file path. This will only affect the channel in which the executing scene is loaded.
PlayScene	Channel.PlayScene(SceneName) **SceneName**: Text	Attempts to play a scene with the specified name or file path. This will only affect the channel in which the executing scene is loaded.
StopScene	Channel.StopScene(SceneName) **SceneName**: Text	Attempts to stop a scene with the specified name or file path. This will only affect the channel in which the executing scene is loaded.

Project: Refers to the parent project of the current scene. Currently this keyword is only used to access parameters defined within the project.

For example, *Project.Parameter1* returns the value of a parameter named *Parameter 1* that exists within the parent project.

Scene: Refers to the current scene.

Item	Usage	Meaning
Layer	Scene.Layer	Returns the layer currently assigned to the scene.
Loaded	Scene.Loaded	Returns a value indicating whether the scene is currently loaded onto a channel.
Playing	Scene.Playing	Returns a value indicating whether the scene is currently playing on a channel.
Name	Scene.Name	Returns the name of the current scene.
Close	Scene.Close	Closes the current scene if it's loaded or playing on a channel.
PlayAction	Scene.PlayAction(Name)	Plays an action with the specified name.
Play	Scene.Play	Plays the current scene if it's loaded; if the scene is already playing then this will have no effect.

Stop	Scene.Stop	Stops the current scene if it's playing; if the scene isn't playing then this will have no effect.

Other Functions

Operator	Usage	Meaning
GetObjectValue	Scene.GetObjectValue(Name, Property) **Name:** Scene object name **Property:** Property name	Returns the value of a specific property defined within a target object. GetObjectValue("Image1", "File")
SetObjectValue	Scene.SetObjectValue(Name, Property, Value) **Name:** Scene object name **Property:** Property name **Value:** Value to assign	Applies a value to a specific property defined within a target object. SetObjectValue("Image1", "File", "C:\\Sample.png")

Expression Evaluation

Only after the entire expression has been evaluated is the result converted to a value appropriate for the given expression type. Consequently, expressions may include mixed data types as long as the result fits the constraints of the expression type. For example:

Expression Type	Expression Text	Evaluation	Value
Double	1 + 7	8	8.0
Integer	1 < (30 – 7)	1 < (23) = True	1
Integer	“10” + “30.2”	“1030.2”	1030
Boolean	(30 < 7) and True	(False) and True	False

Parentheses may also be used to explicitly denote order of operation.

Advanced Bindings

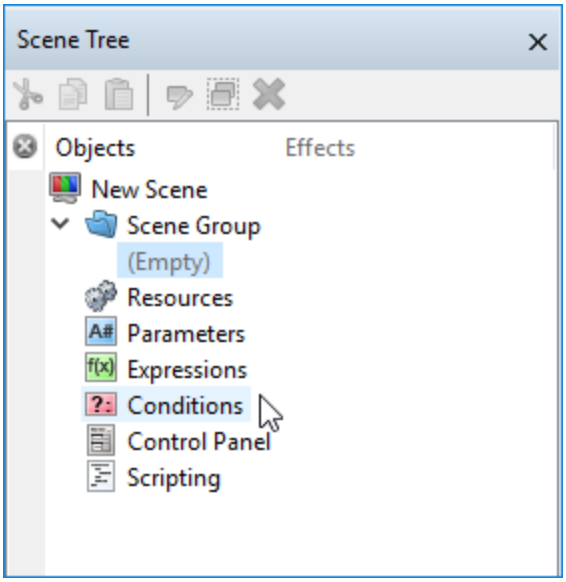
Expressions can also be used when setting up bindings. Instead of simply specifying a target (e.g. **Text1.Text),** the user may use the syntax below to incorporate expressions.
Text1.Text = Expression
Expression is any valid expression text with the additional variable available (VALUE) which evaluates the value of the parameter that has changed. This expression text would be manually entered into the binding column for a parameter or expression. For example, suppose an Integer parameter was configured with the binding expression below:
Text1.Width = value * 10

If the parameter was updated with value 5, then the binding would immediately result in a value of 5 * 10 getting applied to Text1.Width.

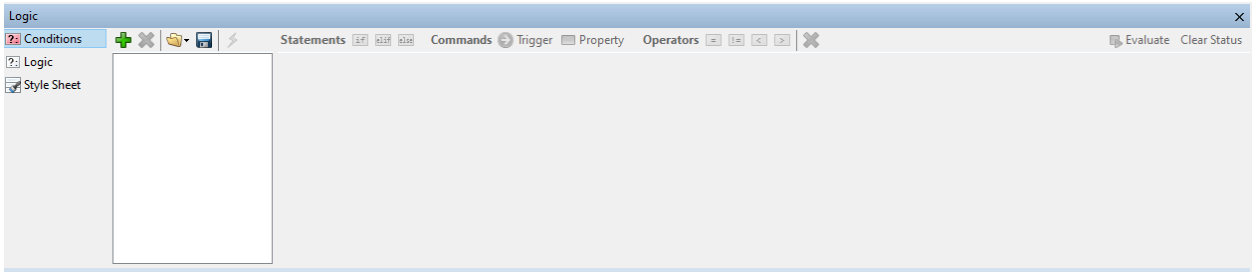
Conditions

Introduction

Condition objects may be utilized to build logic into the execution of event triggers. Conditions appear in the Scene Tree in the same manner as *parameters* and *expressions*, however, a different panel is used to construct and modify them.

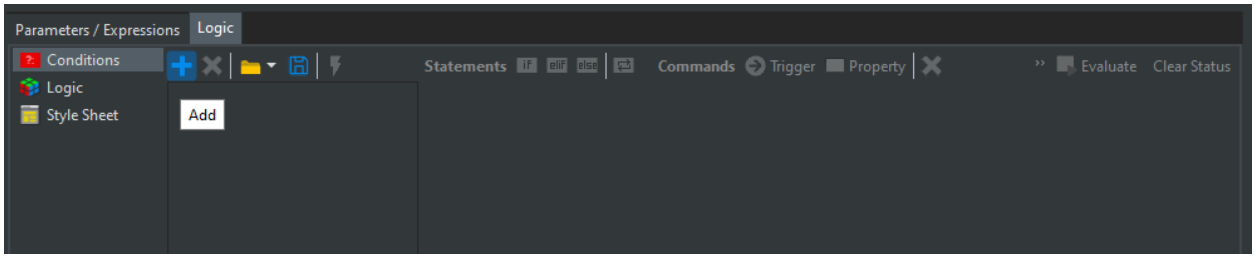


Clicking on the *Conditions* node in the Scene Tree or the Conditions item in the View Menu will display the Logic pane seen below. The Logic pane will display Conditions, Logic effects and Style Sheet effects.

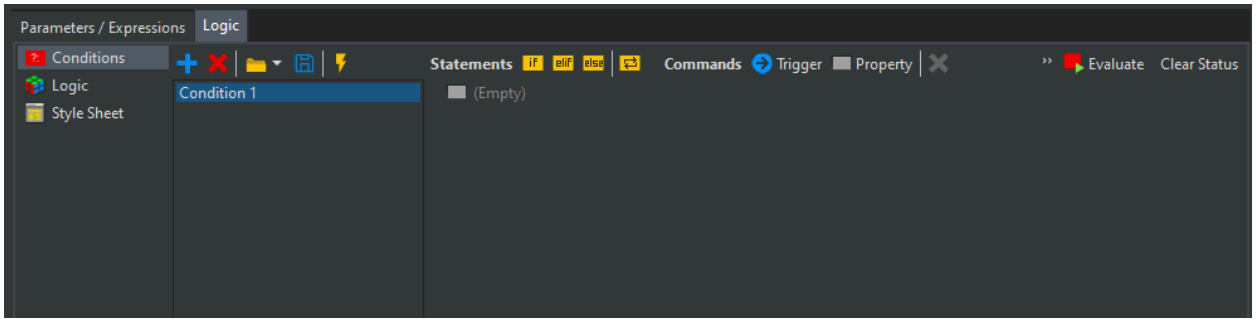


Creating a Condition

The first step to incorporating conditional logic into a scene is to add a new Condition object. This can be achieved by clicking the **Add (+)** button in the editor panel.



This will create a new, empty condition. The Condition may be renamed by clicking the top-level item in the tree. **Names must be unique**.



Once the desired name has been decided upon, the next step is filling out the conditional tree structure. The currently supported conditional statements are:

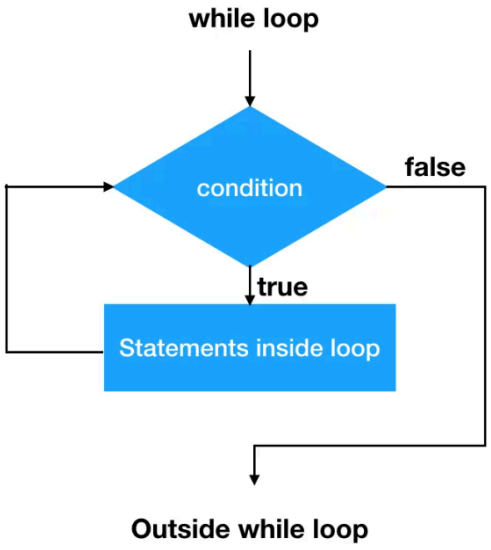
- **If** - This tests a Boolean (*True/False*) expression. If the expression evaluates to *True*, then the statements contained within the **If** will be evaluated as well.
- **Else If** - This tests a Boolean (*True/False*) expression as well, however this statement type may only exist following an **If** statement. If the expression evaluates to *True*, then the statements contained within the **Else If** will be evaluated as well.
- **Else** - This statement type may only exist following an **If** or **Else If** statement. The statements contained within the **Else** will evaluate if the associated **If** and **Else If** statements all evaluated to *False*.

- **While Loop** - In while loop, a condition is evaluated before processing statements inside the loop. If a condition is true then and only then the body of a loop is executed.

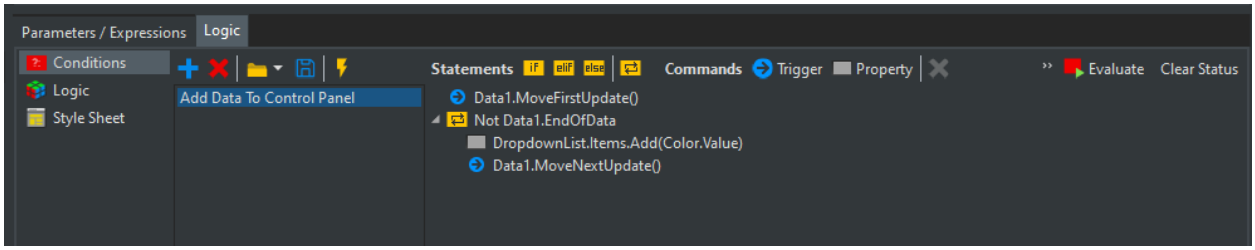
  After the body of a loop is executed then control again goes back to the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false.

  Once the condition becomes false, the control goes out of the loop.

  In a while loop, if the condition is not true, then statements inside the loop will not be executed.



  Example of using a while loop condition would be to loop through a variable quantity of items in a data table column, and have that list of items populate a control panel drop down list.



  Advanced:
  - Timeout: Prevents the while loop from remaining in a continuous loop, and will timeout after the specified duration. This default is 1 second.
  - Asynchronous: Allows all statements within the condition to evaluated at the same, versus being evaluated one at a time
    *Conditions by default run synchronously

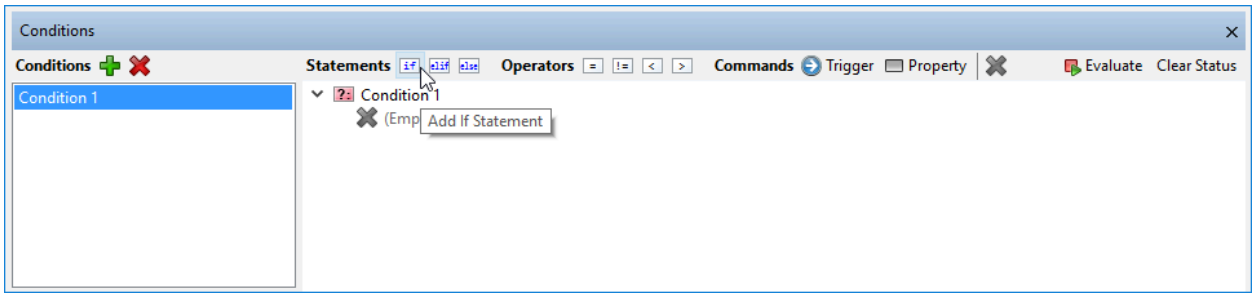
- **Trigger** - This statement type may exist independently of an **If/Else If/Else** test block, or nested within any **If/Else If/Else** item. This defines a list of triggers that will execute when the Trigger statement is evaluated.

- **Property** – This statement type max exist independently of an **If/Else If/Else** item. A property statement is used to modify a specific property of an object using an expression of the form:

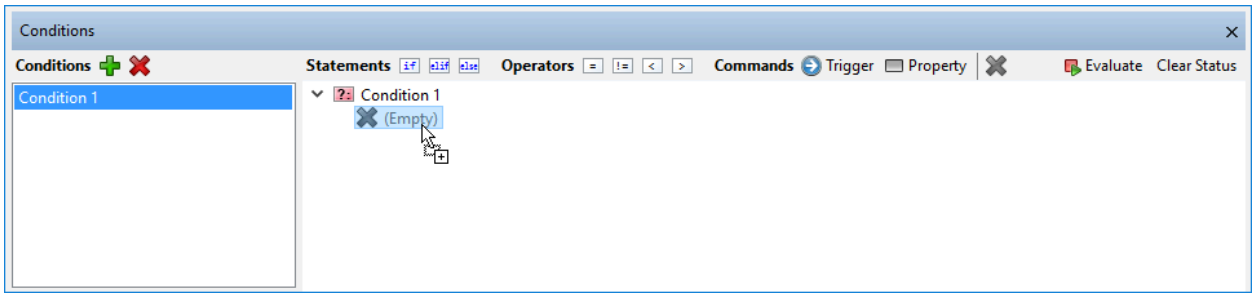
  Object.Property = ExpressionText

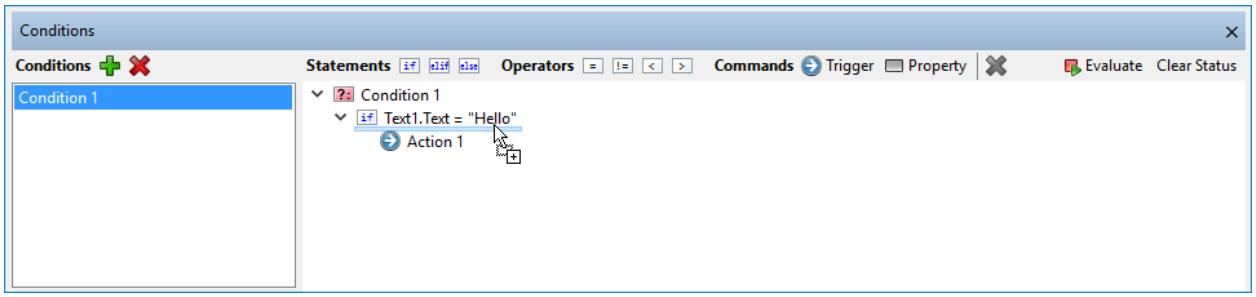
Statements can be inserted by clicking their associated toolbar button or by dragging the toolbar button to the desired insert location.

- Clicking the button will attempt to insert the desired statement at the currently selected location within the condition tree. If the statement is unsupported (e.g. attempting to insert an **Else** statement without an **If** statement), then the insertion request will be ignored.

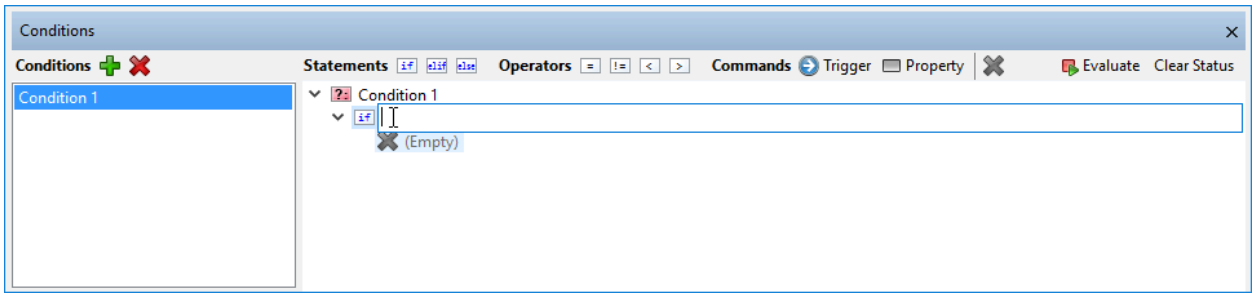


- Similarly, dragging a statement will only allow you to drop at supported locations within the tree (e.g. an **Else** will only be droppable adjacent to an **If** statement that does not already have an **Else** attached).



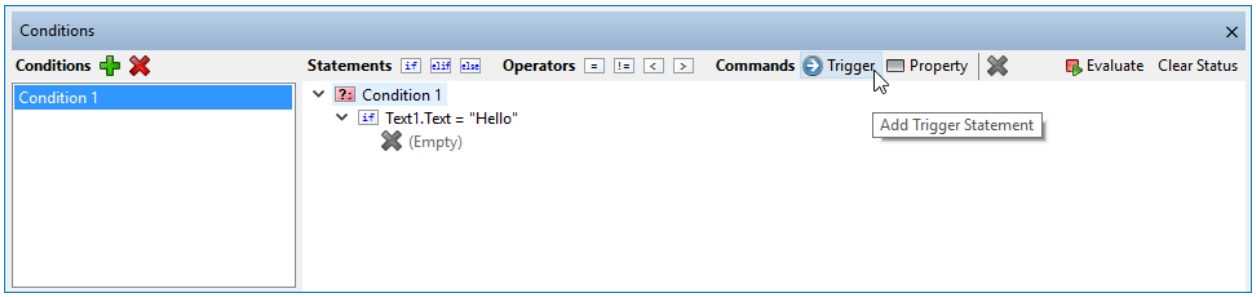


If a statement requires an expression, then an editor will display once it is selected.

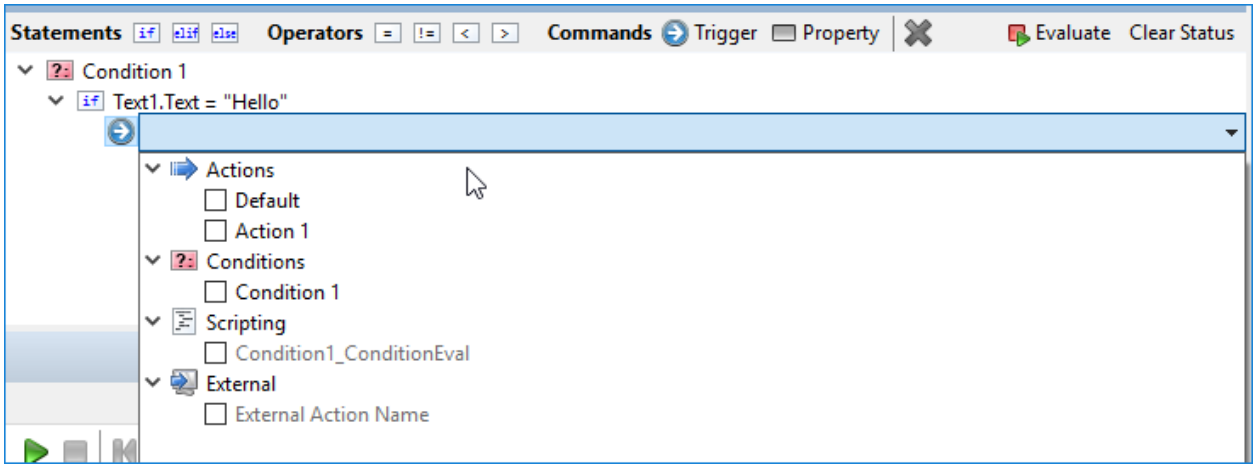


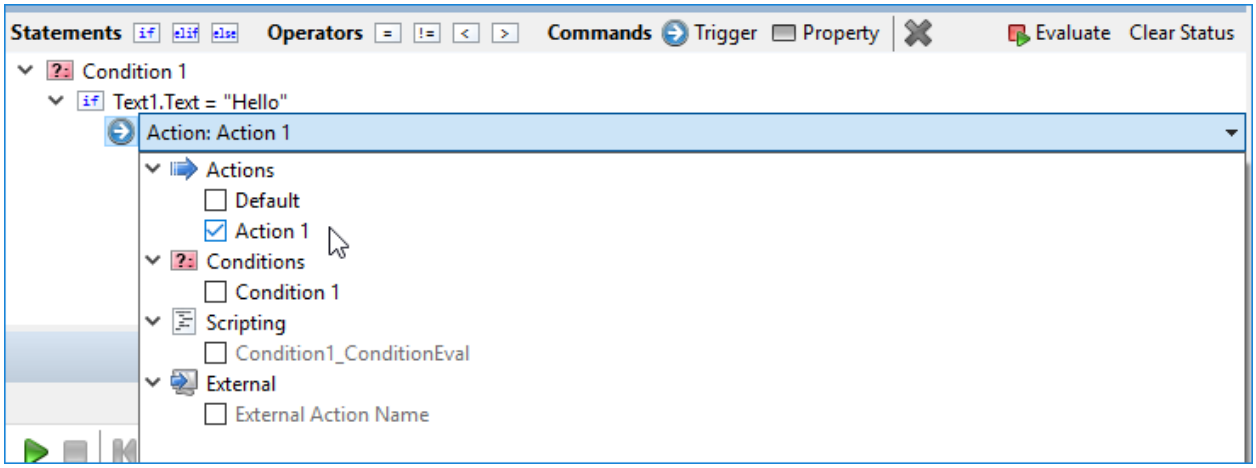
The **If** and **Else If** statements require a valid Boolean expression. For example:

- Text1.PositionX < 100
- Text1.Opacity = Text2.Opacity
- Text1.Text == “ABC”

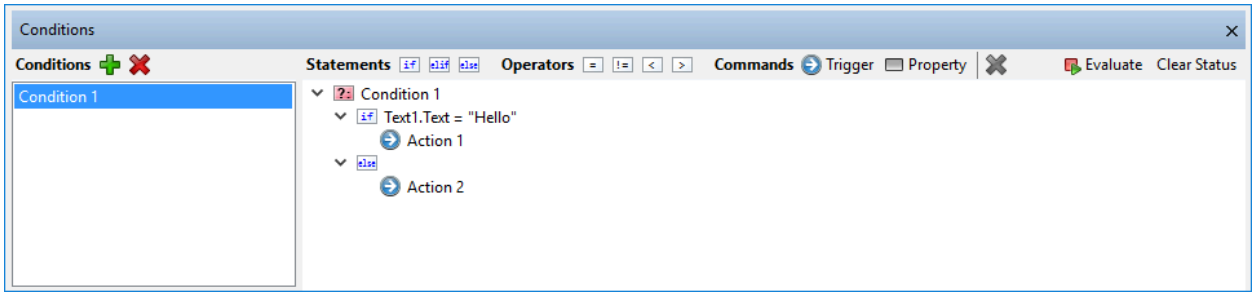


Trigger statements can be inserted independently (outside) **If/Else If/Else** statements or nested inside.



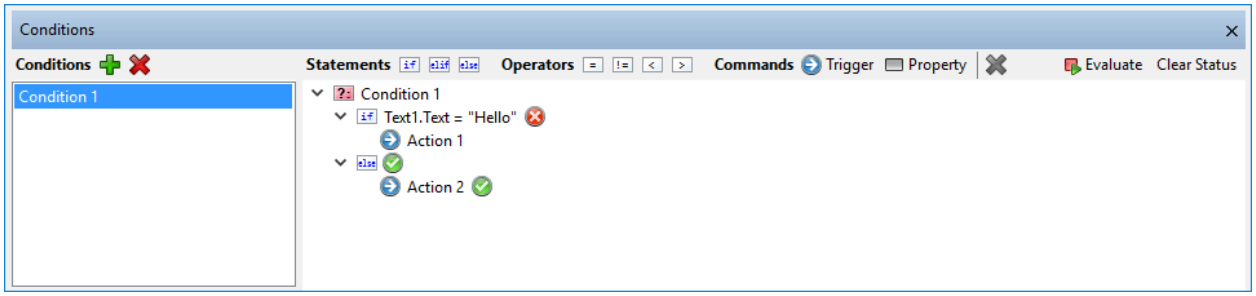


When evaluated, a **Trigger** statement will execute all of the checked triggers in the list. For example:

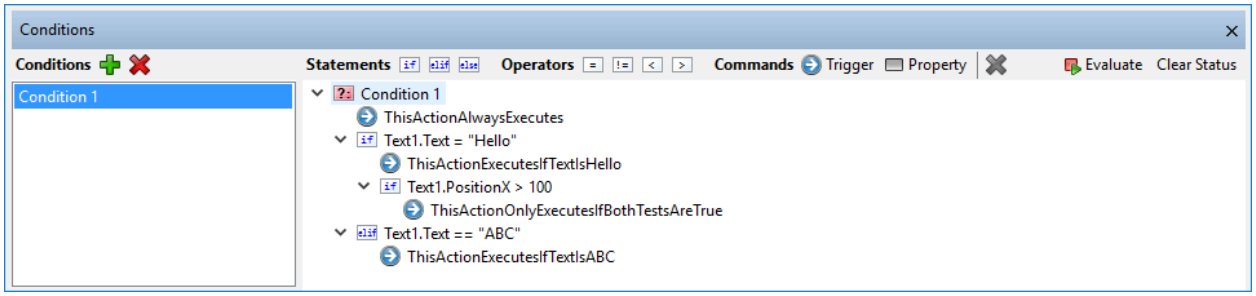


When executing the above condition, Prime will first evaluate whether the Text property of the Text 1 object is currently set to Hello. If the text matches, then Action 1 will be triggered. If the text does not match, then evaluation falls through to the Else statement, which in turn triggers Action 1.

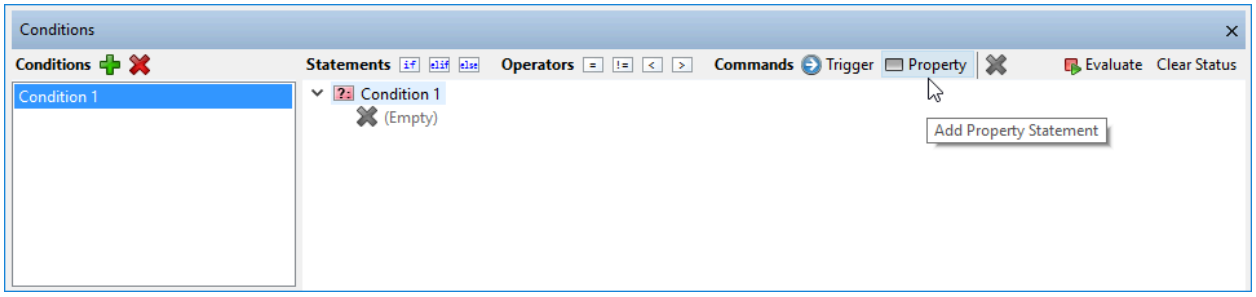
This behavior can be verified by clicking the Evaluate button on the statement toolbar. Icons will appear indicating the evaluation path taken by the condition at the time of execution.



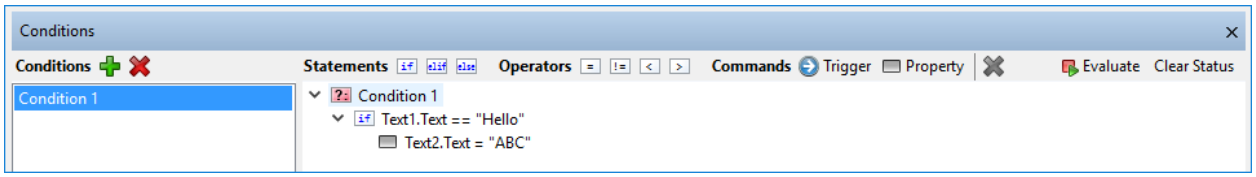
Clicking the Clear Status toolbar button will clear the evaluation icons from the window.



Displayed above is a more complex condition tree.



Property statements may be inserted independently (outside) **If/Else If/Else** statements or nestled inside.



When evaluated, a valid **Property** statement will modify an object property with the result of the defined expression.

Valid **Property** statements have the form:

Object.Property = ExpressionText

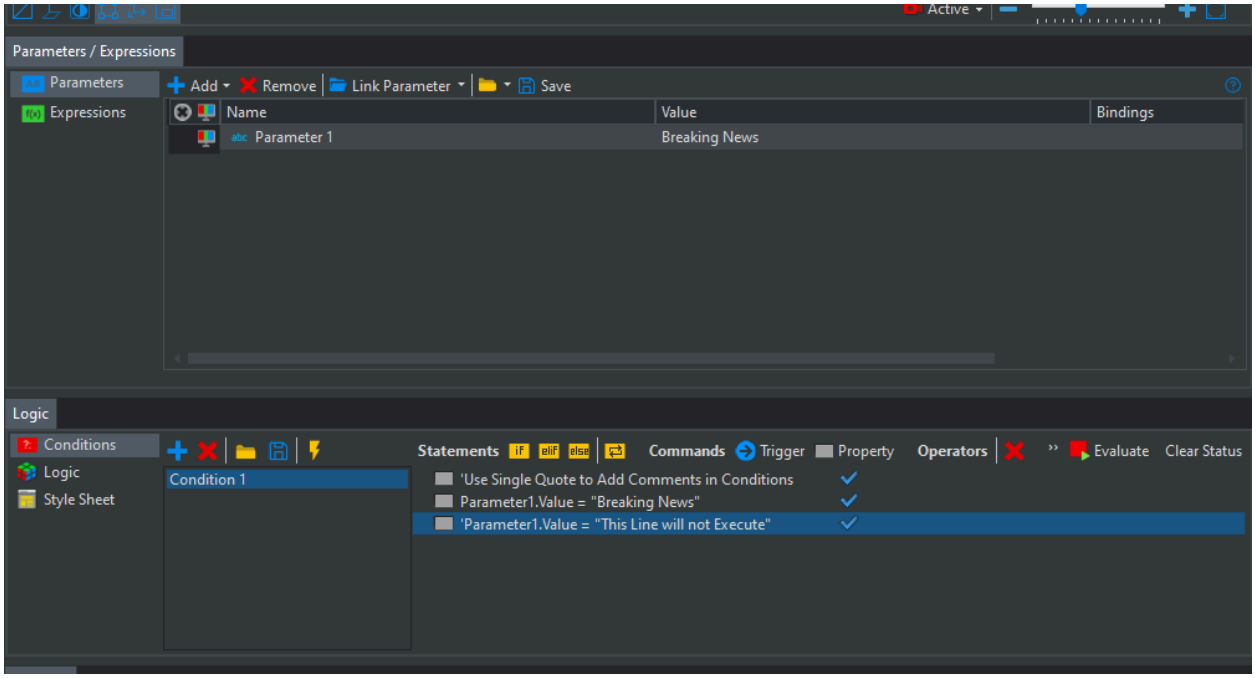
For example:

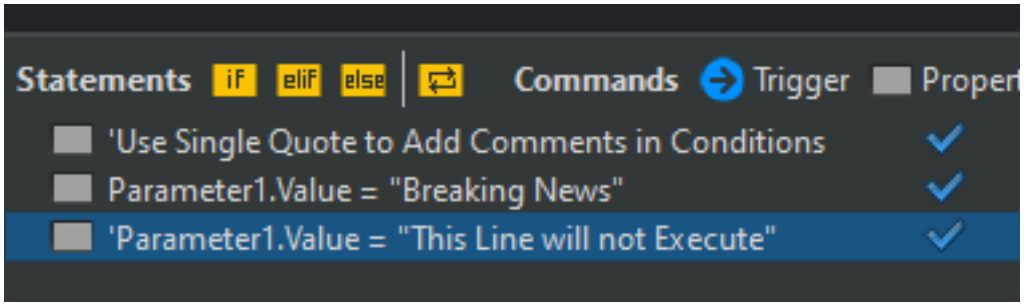
Text1.Test = “ABC” + Text2.Text

In the above **Property** statement, the Text property of Text 1 will be set to the result of adding “ABC” to the Text property of Text 2.

Adding a Comment within a Condition, Logic, or Stylesheets

Typing an apostrophe ('), allows a comment to be added to Conditions, Logic, and Stylesheets which will not be executed. Comments can be used to give context to some logic and comment out logic statements.



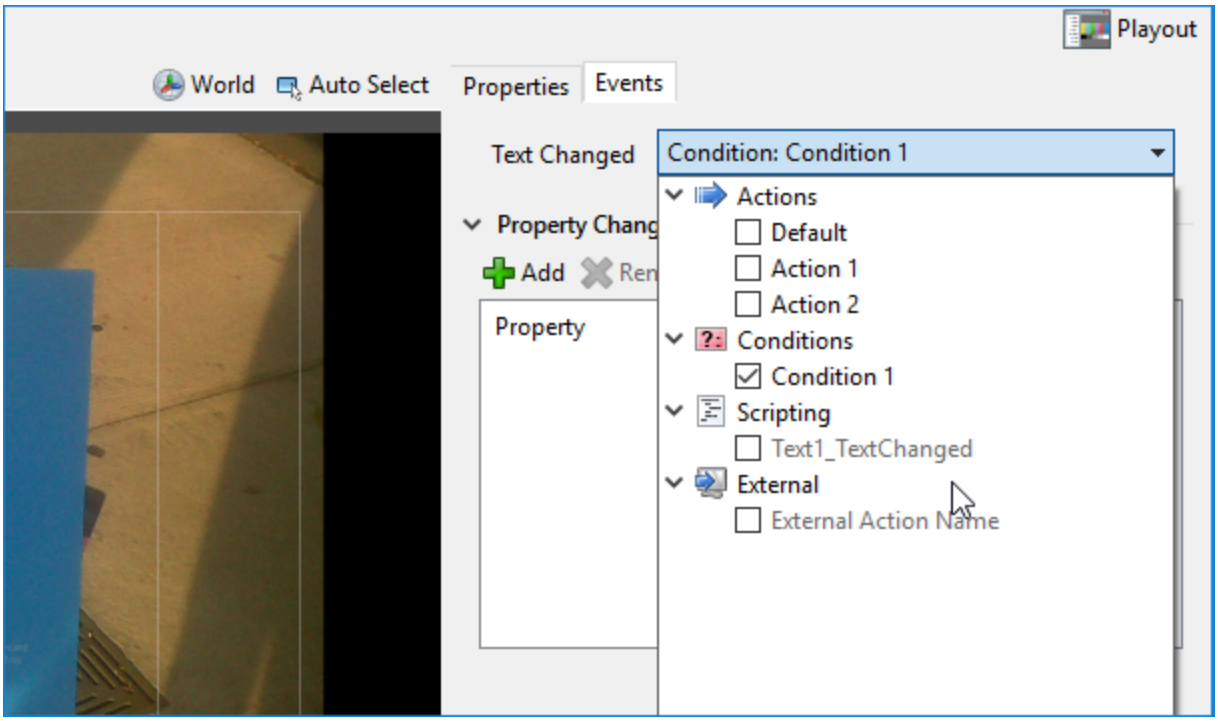


In the example above:

- The first line doesn't have errors as it starts with a quote
- The second line is executed and changes parameter
- The third line is not executed and does not change parameters before it has a single ' in front.

Triggering a Condition

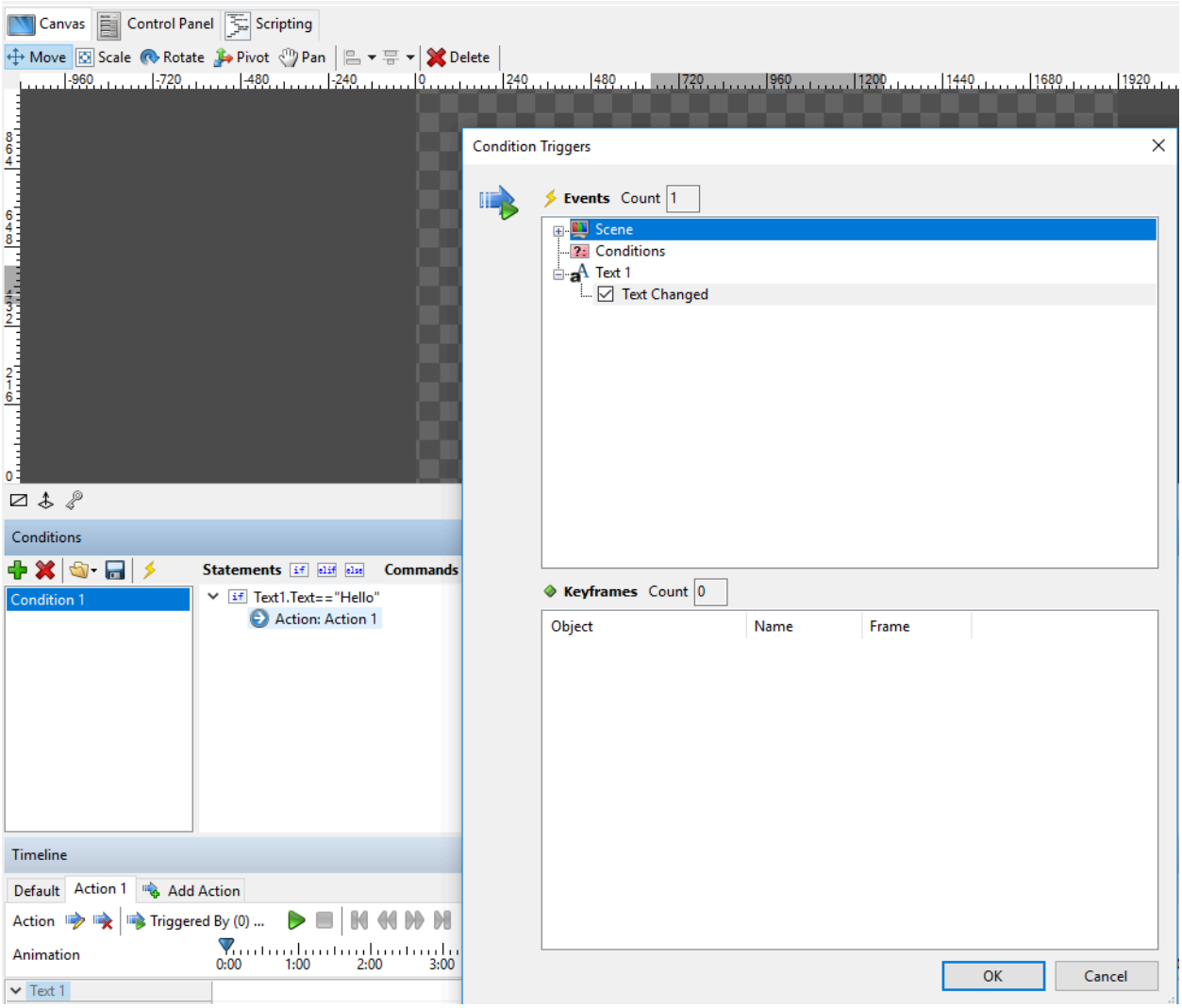
Once the condition object has been configured appropriately, simply choose the condition when editing a trigger list anywhere within the editor. For example, it is possible to execute a condition whenever the Text Changed event of a Text object is raised.



Show what triggers a condition

To determine what triggers a condition, something we called “Triggered By” list click the lightning bolt on the Conditions dialog toolbar. This opens the “Triggers by” list.

This screen shot shows that the “Text1 TextChanged” event triggers Condition1 to execute.



ABOUT US

Chyron is ushering in the next generation of storytelling in the digital age. Founded in 1966, the company pioneered broadcast titling and graphics systems. With a strong foundation built on over 50 years of innovation and efficiency, the name Chyron is synonymous with broadcast graphics. Chyron continues that legacy as a global leader focused on customer-centric broadcast solutions. Today, the company offers production professionals the industry's most comprehensive software portfolio for designing, sharing, and playing live graphics to air with ease. Chyron products are increasingly deployed to empower OTA & OTT workflows and deliver richer, more immersive experiences for audiences and sports fans in the arena, at home, or on the go.

CONTACT SALES

EMEA • North America • Latin America • Asia/Pacific
+1.631.845.2000 • sales@chyron.com

Chyron