GraphWorX64 Advanced Training Manual

Version 10.02

Rev B

© 2012 ICONICS, Inc. All rights reserved worldwide. This document may not be reproduced or copied in any manner without written authorization from ICONICS, Inc.

The information contained within this document is subject to change without notice. ICONICS, Inc. does not guarantee the accuracy of the information.

GENESIS, GENESIS32, Pocket GENESIS, BizViz and their respective modules, OPC-To-The-Core, and Visualize Your Enterprise are trademarks of ICONICS, Inc.

Windows, Windows XP, Windows 2000, Windows 2003 Server, Windows NT, Windows XP-64, Windows 2003-64, Windows Vista, Windows Server 2008, and Windows CE are registered trademarks of Microsoft Corp. in the United States and other countries.

Other product and company names mentioned herein may be trademarks or registered trademarks of their respective owners.

Document Number: TM-GWX64-Advanced-1001

ICONICS LICENSE AGREEMENT

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS!

Opening and using the enclosed software for any purpose indicates your acceptance of the terms and conditions of this license agreement. If you do not agree with them you should return all software, documentation and copy protection keys within seven days of shipment unopened and your money will be refunded.

ICONICS provides this program and licenses its use in the United States, Puerto Rico, or internationally. You assume the responsibility for the selection of the program to achieve your intended results, and for the installation, use and results obtained from this program.

LICENSE

You are granted a personal license to use this program under the terms stated in this Agreement. You may: 1) Install and use the program on a single machine. 2) Make a single archival back-up copy of the program for the sole purpose of supporting your use of the single program on a single machine. 3) You may not use, copy, modify, or transfer the program, or transfer any copy, in whole or in part, except as expressly provided in this license, or with a written contractual agreement with **ICONICS,** Inc.

TERM

The license is effective until terminated. It will terminate if you fail to comply with any term or condition of this Agreement. You agree, upon such termination, to destroy the program and all copies that were made from it and to promptly return any copy protection key(s) to **ICONICS.**

LIMITED WARRANTY

ICONICS WARRANTS THE CD-ROM OR PHYSICAL DISKETTES, ON WHICH THE PROGRAM IS FURNISHED, AND PHYSICAL DOCUMENTATION TO BE FREE OF DEFECTS IN MATERIALS AND WORKMANSHIP UNDER NORMAL USE FOR A PERIOD OF NINETY (90) DAYS FROM THE DATE OF DELIVERY TO YOU AS EVIDENCED BY YOUR RECEIPT. THIS PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

ICONICS SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN PARTICULAR, WITH RESPECT TO ANY PARTICULAR APPLICATION, USE OR PURPOSE, IN NO EVENT SHALL **ICONICS** INC. BE LIABLE FOR ANY OTHER COMMERCIAL DAMAGE, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL OR OTHER DAMAGES. **ICONICS** DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE PROGRAM WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE PROGRAM WILL BE UNINTERRUPTED OR ERROR FREE.

LIMITATION OF REMEDIES

ICONICS' entire liability and your exclusive remedy shall be the replacement of any CD-ROM or diskette not meeting **ICONICS**' "Limited Warranty" which is returned to **ICONICS** with a copy of your receipt within the warranty period. The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages including but not limited to loss of profit, special, incidental, consequential, or other similar claims arising out of the use or inability to use such program even if **ICONICS** has been advised of the possibility of such damages, or for any claim by any other third party.

GENERAL

This Agreement will be governed by the laws of the Commonwealth of Massachusetts. Should you have any questions concerning this Agreement, you may contact **ICONICS** in writing at:

ICONICS, Inc.
100 Foxborough Boulevard
Foxborough, MA 02035

You acknowledge that you have read this agreement, understand it and agree to be bound by its terms and conditions. You further agree that it is the complete and exclusive statement of the Agreement between you and **ICONICS** which supersedes any proposal or prior Agreement, oral or written, and any other communications between you and **ICONICS** relating to the subject matter of the Agreement.

ICONICS RETURN POLICY

All sales are final (NO RETURNS) unless one of the following conditions is applicable: Program is returned in its original packaging material within 7 days after shipment. In such a case, a full refund is provided, less all incurred shipping and handling costs. Program is returned within 30 days from the original shipment date. In such a case a 20% restocking fee is applicable. No returns are allowed on OPC ToolWorX or ActiveX ToolWorX.

Contents

1. Animation and Runtime Menu

1.1 OBJECTIVES 1-1
1.2 INTRODUCTION 1-1
1.3 ANIMATION DYNAMICS 1-1
1.4 ADVANCED RUNTIME MENU 1-15
LAB: OPERATOR HELP MANUAL L1-1
REVIEW: ANIMATIONS AND RUNTIME MENU REVIEW R1-1

2. Smart Symbols

2.1 OBJECTIVES 2-1
2.2 INTRODUCTIONS 2-1
2.3 SMART SYMBOLS IN THE SYMBOLS LIBRARY 2-2
2.4 CREATING A SMART SYMBOL 2-5
2.5 EDITING A SMART SYMBOL 2-8
LAB: SMART SYMBOLS L2-1
REVIEW: SMART SYMBOLS REVIEW R2-1

3. Scripting Basics

3.1 OBJECTIVES 3-1
3.2 INTRODUCTIONS 3-1
3.3 USING THE SCRIPT EDITOR 3-2
3.4 GENERAL SCRIPTING PRACTICES 3-4
3.5 SCRIPTING EXAMPLES 3-8
LAB: BASIC SCRIPTING L3-1
REVIEW: SCRIPTING BASICS REVIEWED R3-1

4. Scripting Advanced

4.1 OBJECTIVES 4-1
4.2 INTRODUCTIONS 4-1
4.3 MANIPULATING OBJECTS IN RUNTIME MODE 4-1
4.4 CREATING A CUSTOM DIALOG USING A FORM 4-6
4.5 READING DATA FROM A DATABASE 4-10
LAB: VIEWING TREND DATA IN A TABLE L4-1
REVIEW: SCRIPTING ADVANCED REVIEWED R4-1

5. Scripting with Controls

5.1 OBJECTIVES 5-1
5.2 INTRODUCTIONS 5-1
5.3 GETTING A REFERENCE TO GENESIS64 CONTROLS 5-2
5.4 WORKING WITH ALARMWORX64 VIEWER 5-6
5.5 WORKING WITH TRENDWORX64 VIEWER 5-12
5.6 WORKING WITH EARTHWORX64 CONTROL 5-19
5.7 CONCLUSIONS 5-32
LAB: SCRIPTING WITH ICONICS' CONTROLS L5-1
REVIEW: SCRIPTING WITH CONTROLS REVIEWED R5-1

A. Glossary A-1

B. Answers B-1

C. Complete Scripts for Labs C-1

Animations and Runtime Menu *1*

1.1 Objectives

After completing this chapter, you should know how to:

- Configure advanced 2D animation properties
- Go to runtime and modify runtime settings

1.2 Introduction

GraphWorX64 is a human-machine interface (HMI) software package for process control where you can create powerful, compelling, and meaningful animated graphics that display real-time data from any OPC compliant data sources. In this chapter, you will learn how to create many different types of animations and how users can customize runtime displays.

1.3 Animation Dynamics

In the GENESIS64 Core class, we briefly introduced you to some of the most commonly used dynamics. As you may have already known by now, a dynamic contains changing information, and you can apply any number of dynamics to an object. In this section, you will learn about the dynamics for making an object animated at runtime.

The types of dynamics you can use to animate an object are listed below:

- The **Size** dynamic changes the size or shape of the object
- The **Location** dynamic changes the location of the object and/or makes the object move along a path that you create
- The **Rotation** dynamic makes the object spin or orbit
- The **Hide** dynamic controls whether the object is visible

- The **Color** dynamic changes the color of the object
- The **State Selector** dynamic displays an object that corresponds to a discrete data value. In the State Selector dynamic, you define all possible states along with the object to display for each state. The State Selector dynamic groups them.
- The **Range Selector** dynamic displays objects in a sequence to create the illusion of an animation; a value provided by the dynamic's data source starts the animation, which can run indefinitely or stop after a specified period of time or number of cycles.

Using these dynamics, we can add animations to the objects on our display. An object on a display can contain multiple dynamics, allowing us to change the look and feel of the object in runtime, as necessary.

Most (but not all) of the above-mentioned dynamics have two animation modes: Analog and Discrete. In Analog mode, the object's animation is always enabled and the duration of the animation is controlled by the data source's value. In Discrete mode, the object's animation can be triggered by a data source value and the duration of the animation is controlled by the time you enter. You will become more familiar with the differences between the two animation modes as you work with them.

1.3.1 Size Dynamic

Use the size dynamic to change the size/shape of the object at runtime. In order to create a size dynamic, you will need to have at least one object on the display. A collection of objects and/or symbols will also work. Once you select an object to which you want to add the dynamic, you can either select the size dynamic from the Dynamics tab of the ribbon or select the Dynamics explorer in the object property window; these buttons are circled in Figure 1.1.

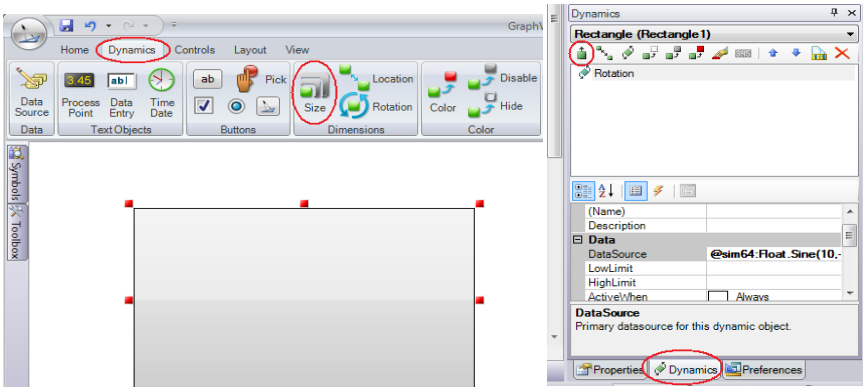


Figure 1.1 - Size Dynamic

Once you add a dynamic to an object, the dynamic automatically appears in the Dynamics explorer when you select the object. Dynamics have many property fields and we will not be able to cover all of them in this course. However, the properties that are essential to proper operation are described here.

> **NOTE**: For more information on individual dynamics and their related properties, you should refer to the GraphWorX64 help files.

Let us first look at the Data section of the Size dynamics properties. There are four fields here:

- DataSource: The OPC point determines when the dynamic is active. In Analog animation mode, this also determines how long it will take the object to grow to its full size.
- LowLimit: The lowest OPC point value that can change the object's size
- HighLimit: The highest OPC point value that can change the object's size
- ActiveWhen: Determines when the dynamic is to be active. You can choose from many different options here. Most of the options use the LowLimit and HighLimit properties.

Use the LowLimit and HighLimit properties to limit the process point values that can change the size of the object. For example, values for the DataSource you have chosen range from 0-100, but you want the object's size to change only for values between 30 and 70, so you set the LowLimit to 30 and the HighLimit to 70.

You may wonder what the difference is between having such limits where the ActiveWhen is set to "Always" and or when you set it between limits. The simplest way to see the difference is for you to try it yourself. You will notice that in such a case, when the ActiveWhen field is set to "Always", you will not see the object at all until it reaches its lower limit and starts growing. On the other hand, when the ActiveWhen field is set to between its limits, when you first go into runtime, you will still see the object in its full size. It will then start to grow when it reaches the lower limit. The behavior difference is caused when the data source is outside of its range limit, causing the animation to be deactivated completely.

In the Dynamic Behavior section, you may want to change the following properties:

- AnimationMode: Analog and Discrete are the two modes of animation
- StartPercent: Percentage of an object that shows when the dynamic starts to animate
- EndPercent: Percentage of an object shown at the end of an animation

- SizeMode: Direction(s) in which you want the object to resize (options shown in Figure 1.2)

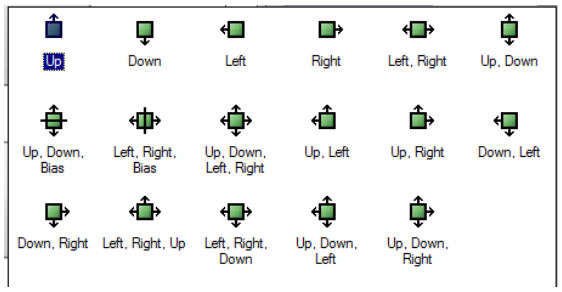


Figure 1.2 - SizeMode Options

If you are using the Discrete animation mode, you may also want to set the following properties in the Dynamic Behavior – Discrete section. These options determine the behavior of the object once its animation cycle begins at runtime:

- StartDelay: Specify the amount of time, in milliseconds, to delay the start of the animation.
- Duration: Specify the amount of time, in milliseconds, to complete one leg of the animation cycle. For example, the animation cycle of an object makes the object grow then shrink; one leg of the cycle makes it grow and the next leg makes it shrink.
- SkipInitialDuration: Set this property to True to skip the first leg of the object's animation and start the object at its end position at runtime startup.
- ReverseAnimateWhenFalse: Set this property to True to create the following behavior: When the data source value evaluates to either False or an inactive state, the animation performs the *opposite* of what it would do were the data source value to evaluate to True. Because this is the size dynamic, the object shrinks instead of grows.
- AutoReverse: Set this property to True to reverse the animation when the object reaches the end of its leg. For the size dynamic, setting this property to True has the same effect as setting the data source to a sine tag in analog mode. If you set it to False, the object disappears or stops, depending on the value of the dynamic's other properties.
- RepeatCount: Specify the number of times the animation cycle is to repeat before stopping. You can choose Infinity if you wish.

1.3.2 Location Dynamic

Creating a location dynamic is similar to how you create a size dynamic. Once you add a location dynamic to an object, you will notice a gray opaque copy of the object in the display, and a red line attaches the object to its opaque copy.

The opaque copy of the object marks the ending point of its animation path; you can move the opaque copy to wherever you want the path to end. To change the ending location of the object's animation, click the opaque copy, which changes it from gray to red (the *selected* object). Then drag it to wherever you want the object's end position to be.

You can also create the path that the object will travel at runtime when the object becomes animated. To create the object's path, click anywhere on the path line (the line between the object and its opaque copy); a handle will appear on the path line. Click the handle and drag the line to change the path. Continue doing this along the path until you have created the path you want. With this flexibility, you create a unique path where the object will travel at runtime, like the path shown in Figure 1.3.

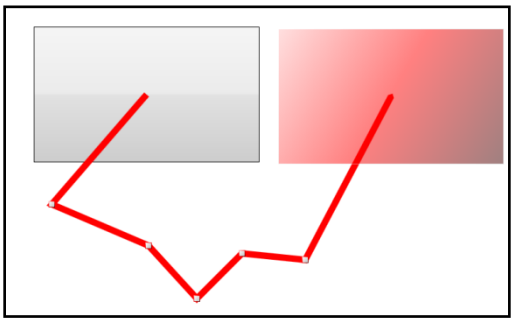


Figure 1.3 - Location Dynamic Flight Path and End Location

The location dynamic's properties are similar to the size dynamic's properties. We will not go over those properties again since they work in the same way except for a few properties that are unique to the location dynamic, which are:

- Slider: This option is only available in analog mode. You can set a location dynamic to be a slider by setting this property to True. This will allow you to move the object left and right, or up and down, to create a slider.
- AutoRotate: Setting this property to True, rotates the object when its animation path reverses. This is useful if the direction that the object faces is important. For example, if the object were a truck, setting this property

to True would rotate the truck when it reaches the end of its animated path so that the cab of the truck always faces forward along its return route.

- Path: This property defines the coordinates of the object's path from the object's start to its end location. Clicking the ellipse button shows the object's path line and its opaque copy.

1.3.3 Rotation Dynamic

Creating a rotation dynamic is similar to how you create a size dynamic. Once you select the rotation dynamic, you will see the object turn red and a small circle will appear in the middle of the object. This small circle is the pivot point, which is the point around which the object will rotate.

You can change the location of the pivot point by clicking it, then holding down the left mouse button and dragging it anywhere you would like to make the pivot of the object, even outside of the object itself. After you change the pivot point, you can see that the PivotOffsetX and PivotOffsetY values have changed as well.

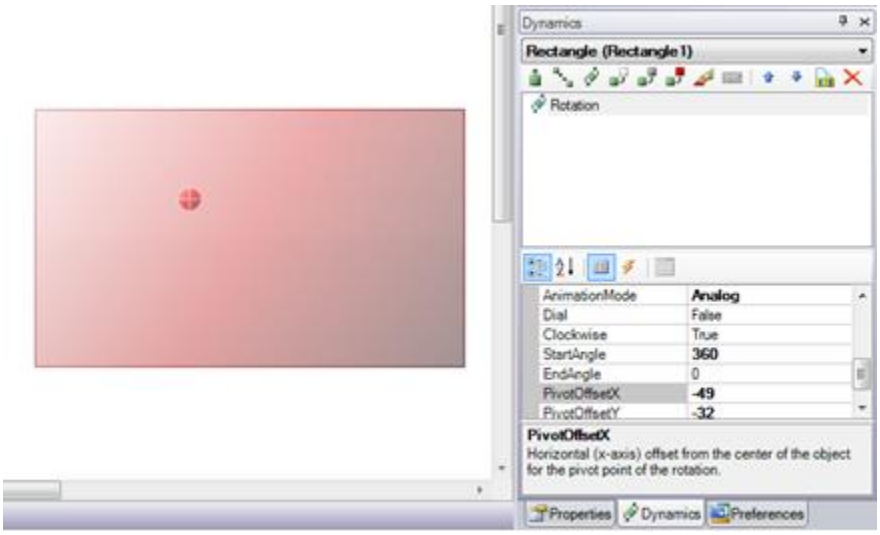


Figure 1.4 - Rotation Dynamic and Pivot Point

Let us look at the rotation dynamic properties now. Again, we will only describe those that are unique to the rotation dynamic.

- Dial: This option is available only in the analog animation mode. If you set this property to True, the rotation dynamic will act like the needle of a dial where the needle points to values on the dial.
- Clockwise: Determines the direction of rotation. If you set it to True, the object will rotate in a clockwise direction; if you set it to False, the object will rotate counterclockwise.

- StartAngle: Determines the angle position at which the object starts its rotation. Valid angle values start at 0 and go to 360. Your pivot point and the shape of your object determine Angle 0. In general, if your object is longer on the Y-axis, your 0 angle is north; but if your object is longer on the X-axis, your 0 angle is east.
- EndAngle: Determines the angle position at which the object's rotation ends. This property works the same way the StartAngle works.
- PivotOffsetX: The number of units to move the pivot point away from its original position along the X-axis.
- PivotOffsetY: The number of units to move the pivot point away from its original position along the Y-axis.

1.3.4 Hide Dynamic

The hide dynamic, as its name suggests, allows you to make the object appear or disappear based on the value of the data source. There are only two states for this dynamic: show or hide. For that reason, the data source you use for a hide dynamic is usually a digital tag. The animation mode is set to Discrete by default.

> **NOTE**: Because the nature of the Hide and Color dynamics, you should never set these dynamics to analog animation mode.

There are no other properties. However, it is worth mentioning that the hide dynamic's Duration property determines how long it takes the object to fade in and out. The Duration property does not determine how long a time the object shows and hides; instead, the data source value determines how long at a time the object shows or hides.

1.3.5 Color Dynamic

The color dynamic changes the color of the object based on the data source value. Like the hide dynamic, the color dynamic uses discrete animation mode by default because, in most cases, you will want the color to change in response to a change in a tag value.

The color dynamic's essential properties are listed below:

- UseStartColor: False is the default. If you set the property to True, you can also select a starting color for the object. If you do not enter a Duration, the color you choose here overwrites the object's color in runtime. If you do enter a Duration, when the dynamic is activated, the object will fade from the start color to the end color in the duration time.

- StartColor: Choose the object's starting color.
- UseEndColor: This is set to True. Specify the color the object will change to when the ActiveWhen property is True.
- TargetPropertyName: Specify whether you want the line of the object (the outside edge) or the fill of the object to change color.
- TargetPropertyIndex: Specify the color range. Use this property if you want a gradient for the changed color. For example, if the original fill of the object has two colors, you will need to add two color dynamics and set the end color of each dynamic to match the color you want for the gradient. Then, in this property, you indicate which color goes where on the gradient. The index starts at 0.
- FlashRate: Makes the object flash or blink when the dynamic is active. Enter a value in milliseconds as the flash rate.

If you want to associate multiple color changes to the object based on the data source value, set the ActiveWhen dynamic to Data Equals Low and set the LowLimit to the value at which you want the object to change color. Of course, the same idea can apply to a range of values changing to the same color.

1.3.6 Selectors

Selectors are dynamic actions that you apply to a group of objects so that, at runtime, different objects display at different times based on the criteria you specify and/or the data values sent from the data source.

As you work with selectors, you may notice that the dynamics are only available when you select two or more objects. That is because you need multiple objects (different states) to make a selector. You cannot simply apply a selector dynamic to a single grouped object, but you can select the objects within a group and make it a selector. If the selected objects are not already in a group when you apply the dynamic, GraphWorX64 will group it for you, and you cannot ungroup the object without removing the dynamic.

1.3.6.1 State Selector Dynamic

A State Selector dynamic is an object that is essentially ***a switch***. It changes from state to state, and each state is driven by the data sent to the dynamic from a data source such as an OPC tag on sensor or a user-entry field on a dashboard. You might use a State Selector for a traffic light (three colored circles in a group); a series of signs such as Stop, Go, and Yield; or an on/off switch. To remind you of its uses, the State Selector icon has a switch on it.

Figure 1.5 - State Selector Icon

A State Selector dynamic determines which graphical object to display as determined by the data source's current value or state. Use this type of dynamic to show graphical objects to depict individual states. For example, you might use this dynamic for a traffic light, which has three states, each represented by a color (green, red, or yellow).

We will demonstrate how you might create an automated traffic light using the State Selector dynamic. First, you will need to create three objects, each containing a different state of the traffic light, as shown in Figure 1.6. Notice that there are three different grouped objects in the Explorer, one for each state of the traffic light.

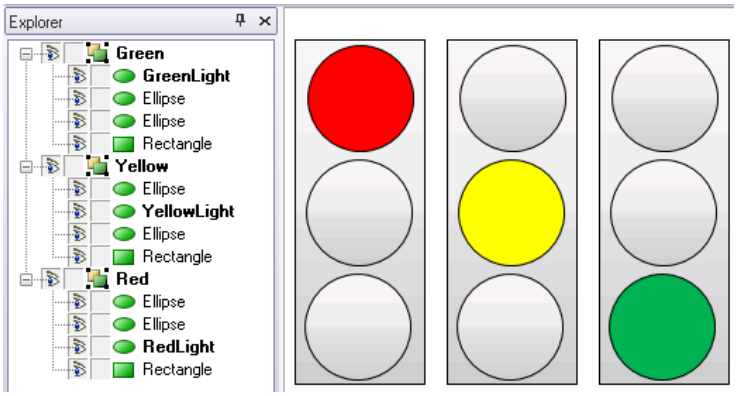


Figure 1.6 - Three Traffic Light Objects Used for Creating the State Selector Dynamic

Then, move the three objects layered on top of each other. You can achieve this simply by aligning the top and left of all three objects. With all three objects selected, click the State button in the Selectors section of the Dynamics tab of the ribbon.

Clicking the State button groups the three traffic light objects into a State Selector dynamic, which is shown in Figure 1.7.

Figure 1.7 - The State Selector (Dynamic Group is selected in the Explorer)

Set the dynamic's **DataSource** property by clicking the DataSource ellipsis button to open the **Data Browser**. You either create or select a simulated data source such as **localsim::triangle** or choose a real OPC tag.

The dynamics **States** property is a collection of the different states that you define on what to show for a certain value. You can bring up the **Edit Selector States** dialog box, which is shown in Figure 1.8, by clicking the ellipsis button where it says **(Collection)**.

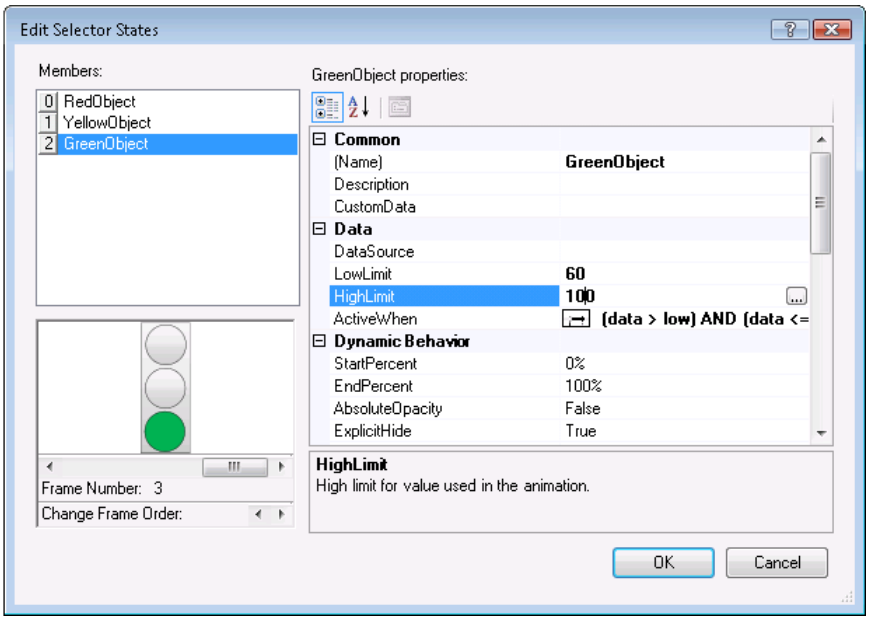


Figure 1.8 - The Green Object's Edit Selector State Settings

In the Members section of the dialog box, you can see the different objects that you selected to create this dynamic. Below that is a screen that shows you the actual objects. You can click the scroll bar to see the order that the object would appear in. If you want to change the order of appearance, you can use the arrows in the Change Frame Order section. Clicking the left arrow moves the current frame to the previous one and clicking the right arrow moves it to the one after. It may take some practice to get all the frames in the correct order.

The most important parts are the properties for the individual objects are in the right of the dialog box. The properties are similar to those in the Properties explorer in your GraphWorX64 container. For object in the dynamic, you should at least configure the Data section. This section controls when the particular object (or frame) would appear.

Assuming that we are using the **localsim::triangle** tag and we want to show the red light when the value is between 0 and 40, the yellow light when the value is

between 41 and 60, and the green light when the value is between 61 and 100, we would set the properties for the three states as shown in Table 1.1.

Table 1.1 - Data Configuration for Each Member Object

Member Object	LowLimit	HighLimit	ActiveWhen
RedObject	0	40	(data>=low) AND (data<=high)
YellowObject	40	60	(data>low) AND (data <=high)
GreenObject	60	100	(data>low) AND (data <=high)

Figure 1.8 shows the green state and its definition in the Edit Selector States dialog box. Note that we do not need to set the DataSource for each object because it inherits the DataSource we gave to the dynamic. Of course, if the occasion calls for a different data source, you can overwrite it per each object.

Once you are done configuring the object, you can click the Runtime button to watch the State Selector dynamic in runtime mode. If you want to see the actual values from the data source, add a Process Point on to the display, and use the same data source that you used for the State dynamic. Figure 1.9 shows a time-lapsed example of the traffic light.

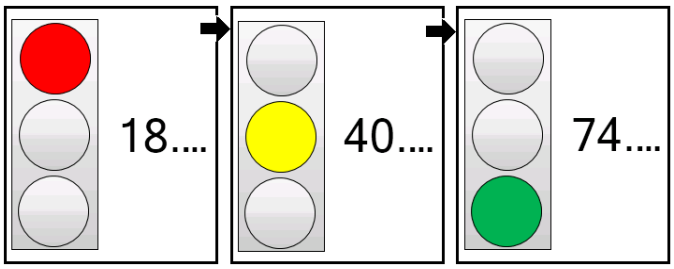


Figure 1.9 - Traffic Light in Runtime Mode

1.3.6.2 Range Selector Dynamic

A Range Selector dynamic is an object that animates a group of selected objects in a sequence of frames so that the objects play much like a movie. You might use a Range Selector dynamic to start an animation of a waving flag when the day's quota is met. Or you might use a Range Selector dynamic to set in motion an animation of a proper installation of repair technique.

Figure 1.10 - Range Selector Icon

A Range Selector dynamic triggers an animation when the data source's value or state reaches a predetermined value or state. The animation consists of a series of

graphical objects (which, in this context, are also called ***frames***) that display in sequence. It is quite similar to the State Selector. If you are familiar with the analog and discrete animation modes, you can consider the State selector the analog mode of the animation and the Range selector the discrete mode.

Before you can begin a Range dynamic creation, you should add to the GraphWorX64 display ***all*** of the frames that make up the animation because once you create the dynamic, you cannot add or remove frames. Each frame in the animation is one object. Hence, if there are fifteen frames that make up the animation, make sure that you add each of the fifteen frames to the display. Figure 1.11 shows four simple objects that make up four frames of a small animation.

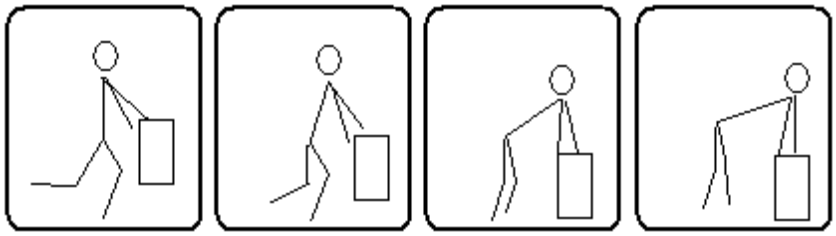


Figure 1.11 - Four Objects for Use in a Range Selection Dynamic

In the display, place each frame in the location where you want it to appear during the animation. The objects can lie directly on top of each other so that they play in one spot, or they can be placed so that when the animation plays the subject appears to be moving around. How you position them depends on their content and the animation that you want to achieve. However, you can reposition the frames in the display at any time.

> ***Tip!*** When naming the objects in the Explorer, include numbers in the objects' name so that you know their sequence in the animation. Sequence the objects in the Explorer in the order in which they are to display, putting the first frame at the bottom and the last on top. This way, when you create the selector, it sequences the frames in the correct order and you do not have to change it later.

When all of the frames are included in the display, you can create the Range Selector dynamic by first selecting all of the objects (that is, all of the frames) that make up the animation. With all the desired objects selected, click the **Range** icon in the **Selectors** section of the **Dynamics** tab on the ribbon. This puts the frames into a group that displays in the Explorer, and creates a Range Selector dynamic for the group. Figure 1.12 shows some of the properties for the Range Selector, which you will need to configure once you create the dynamic.

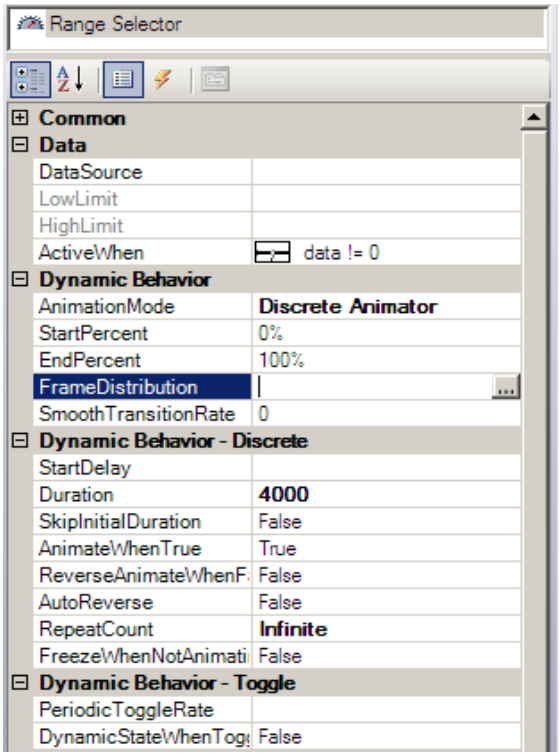


Figure 1.12 - Range Selector Properties

Since this is a "discrete" Selector dynamic, we will need a data source that controls the animation. Go to the Range Selector dynamic's DataSource property and click the ellipsis button, which opens the Data Browser, and then select the data source that will trigger the animation to begin. Next, you should give a condition in the ActiveWhen property. When the data source meets this condition, the animation will start; if not, it will stop.

In the **Duration** property, specify how long, in milliseconds, you want it to take the animation to complete one cycle from beginning to end. For example, if you want the animation to complete the cycle in one second, enter 1000, if you want it to complete the cycle in 3 seconds, enter 3000.

In the **RepeatCount** property, specify how many times you want the animation cycle to complete before stopping. If you do not want it to stop, select **Infinite**.

> ***Tip!*** The AnimationMode property's default value is Discrete Animator, although you can change it to be an Analog Animator, it does not make sense to do it. You should leave this property with its default value.

Clicking the **FrameDistribution** property's ellipsis button opens the **Edit Frame Distribution** dialog box. Below the display area is a slider, which allows you to

page through each frame in animation. Click in the slider area to page to a different frame. You can click the left and right arrows on the **Change Frame Order** bar to move the current frame to a different spot in the animation sequence (this is highly similar to the frame editor in the State Selector). You can use the **Selected Range** sliders at the bottom of the dialog box to adjust the length of time spent on any individual frame. Once you are done, click OK to save the changes and exit the dialog box.

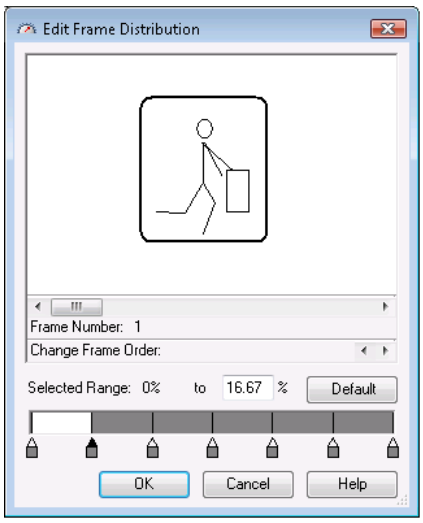


Figure 1.13 - Editing frames for the Range Selection dynamic

You can use the **SmoothTransitionRate** property to add a fading element to each frame in the animation. For this property, you can specify how many milliseconds you want it to take objects to fade away. Zero (0) milliseconds, which is the default, uses no fading at all; typing 3000 milliseconds (or 3 seconds) means that each frame will fade out over three seconds.

Use the **PeriodicToggleRate** property to start and stop the animation. By default, once the animation begins it will run as long as the Range Selector dynamic's ActiveWhen property resolves to true. However, you can use the **PeriodicToggleRate** property to toggle the animation on and off by specifying a number of milliseconds or by choosing a data source that will toggle it. For example, if you enter 5000 milliseconds, the animation will run for 5 seconds, then stop for 5 seconds, then run for 5 seconds, and so on. Or you can specify a data source that will stop and start the animation.

Once you are done configuring, click the **Runtime** button to go into runtime mode and test your Range Selector.

1.4 Advanced Runtime Menu

In runtime, users can employ the runtime menu bar to control how they view displays. It is limited in comparison to what the configuration mode offers, but it still gives the user flexibility to do the things that they need to do most of the time. Figure 1.14 shows the runtime menu with added inline explanations.

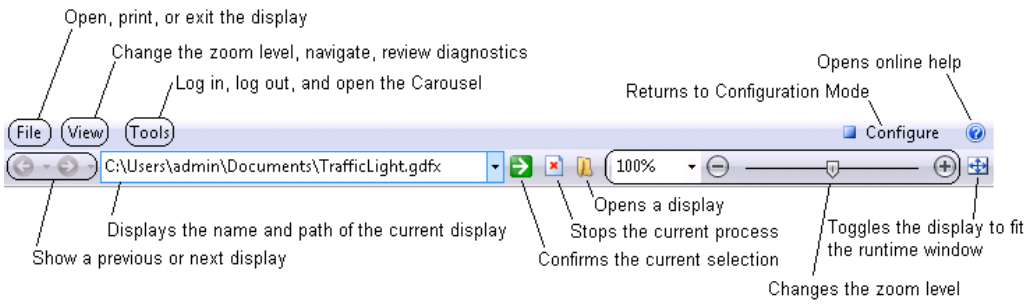


Figure 1.14 - Advanced Runtime Menu

1.4.1 File Menu

The File menu, shown in Figure 1.15, provides a user the possibility to open another display and print the currently opened display. Users may also quit GraphWorX64 if they have the permissions to perform this action.

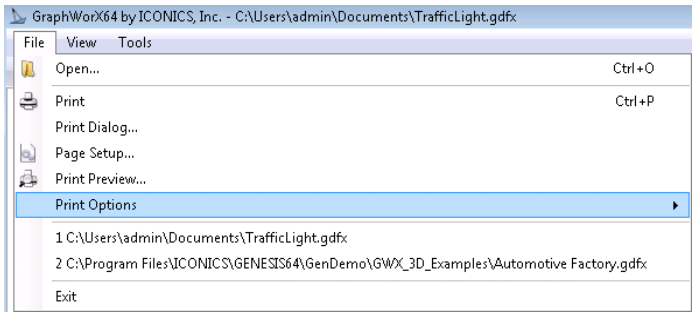


Figure 1.15 - File menu at runtime

1.4.2 View Menu

In the View menu, you can find different view modes that give users the ability to zoom in/out on the display. Users are also able to show/hide certain navigation menu or bars from here and modify what users see during Runtime. Besides changing icon size and being able to go back and forth between displays, users can open the Diagnostics window.

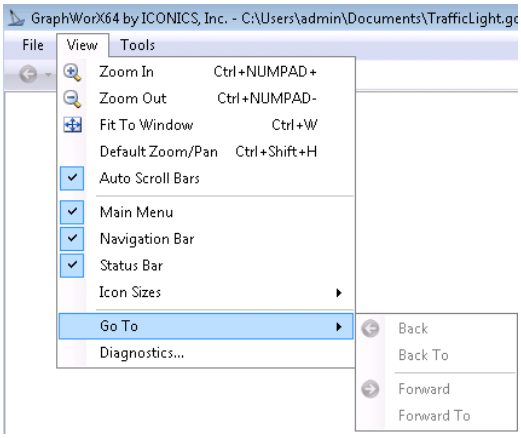


Figure 1.16 - View Menu at Runtime

The Diagnostics window, shown in Figure 1.17, lists all tags and expression that have errors in the current display. Instead of hovering your mouse over the tag that is not giving you good quality data, you can simply go to the diagnostics window and see the list of all the bad quality tags or expressions. Good quality data sources will not be listed in this diagnostics window.

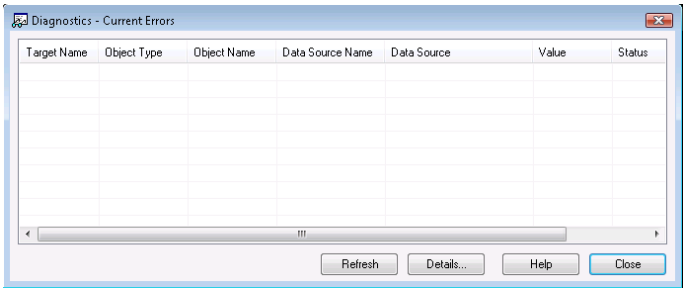


Figure 1.17 - The Diagnostics Window at Runtime

1.4.3 Tools Menu

The Tools Menu allows users to log in and out if there is any security configuration set up. This menu gives users the opportunity to select the language, and an option to run Carousel, usually for demonstration or testing/debugging purposes.

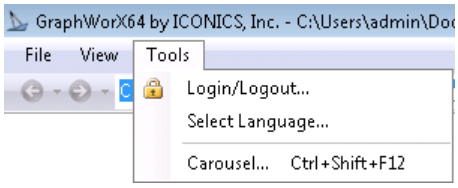


Figure 1.18 - Runtime Tools Menu

Lab: Operator Help Manual *1*

1.1 Lab Overview

Now that we have provided icTonics a great set of graphics for their operators to control the bottling process, they would like us to create a screen that shows simple diagnostic steps to help the operators resolve the issue. The screen should include some common problems and the solutions to it as well as contact information to the engineers in case they were not able to resolve the issue following the steps.

Estimated time to complete this lab: 20 minutes

1.1.1 Task Checklist

In this lab, you will complete the following tasks:

- Create different screens for help instructions
- Make the help screens a walk through using the State Selector dynamic

1.1.2 Support Files

You will need the following files to complete this lab:

- icTonics US Facility.gdfx

1.2 Creating a Help Screen

We will use a state selector in this lab to create a guide that helps an operator resolve some issues that may regularly happen.

1. Open your icTonics US Facility.gdfx display and add a help button.

2. This button should perform a Load Display command that targets a Popup Window (Display Dependent) type window. The display that it opens should be called icTonicsHelp.gdfx.

3. Open a new display and apply the icTonics Template to it. Remember to save your display periodically throughout this lab.

4. On this display, create a rectangle that's 974 x 625 with the left at 25 and the top at 118.

5. You can also change the background color to white.

6. As you add a few objects to the display to make a page that is similar to Figure L1.1, set "stretch: none" and "font" to 14 on the right-hand side tool bar.

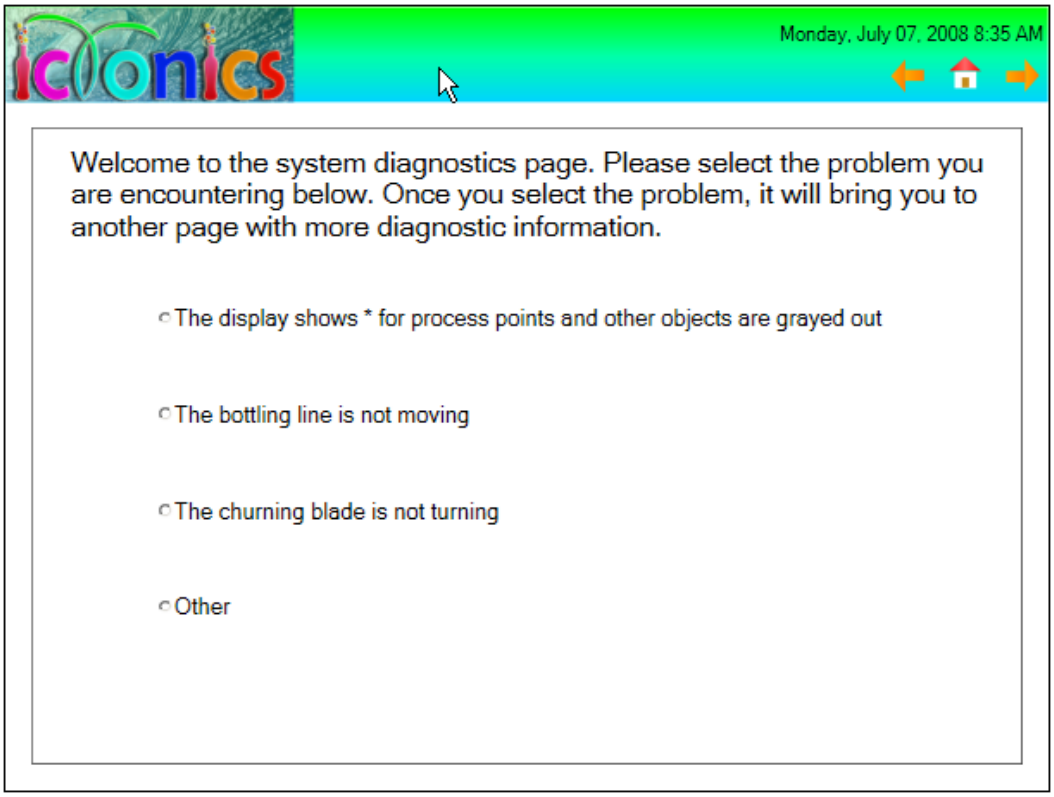


Figure L1.1 - First Help Page

7. The radio buttons on the display should use the default Write Value command. The value should be written to a local simulation variable. Each of the buttons should write to the same variable. The first radio button should write 1 to the variable, the second one should write 2 to it, and so on.

8. Group these objects together and call it Screen0.

9. Duplicate Screen0 and name the duplicate Screen1. Align the screens so that one is on top of the other.

10. Configure Screen1 to look similar to Figure L1.2.

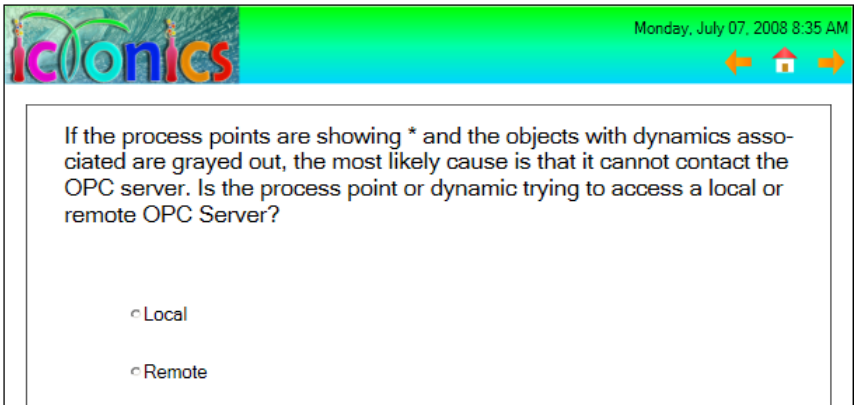


Figure L1.2 - Screen1

11. The first radio button should write a value of 5 and the second a value of 6 to the simulation variable.

12. Duplicate the screen again and call the duplicate Screen2. Align the screens again.

13. Make the screen look similar to Figure L1.3.

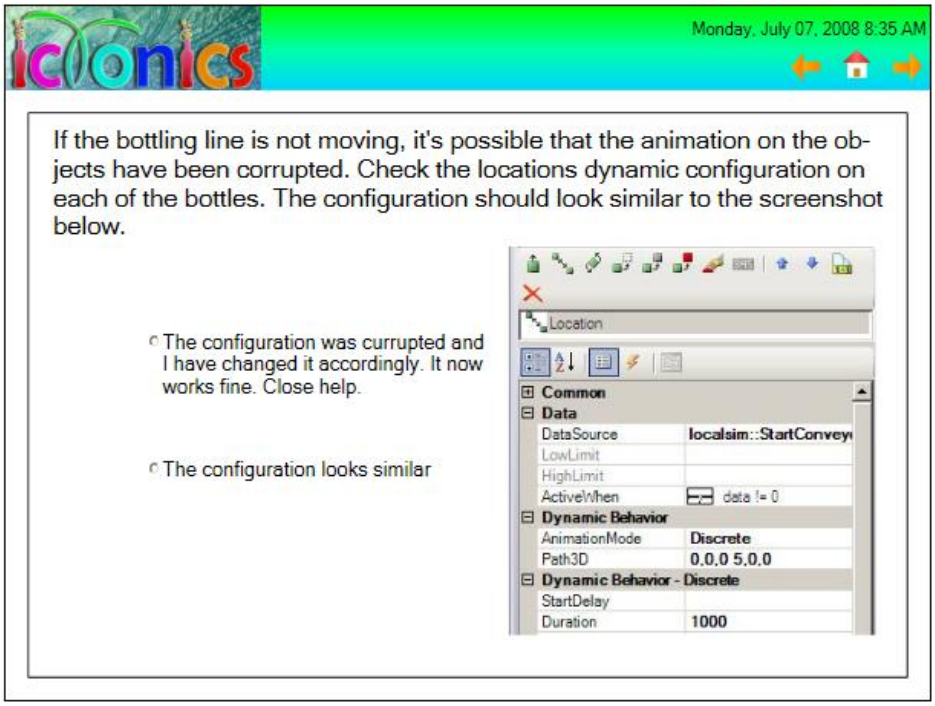


Figure L1.3 - Screen2

14. The First radio button should perform a close window command while the second radio button should write a value 4 to the variable. You can insert the screenshot by opening your icTonics US Facility display and finding the property to the bottle.

15. Repeat step 11, and name the group Screen3. Make Screen3 look similar to Figure L1.4.

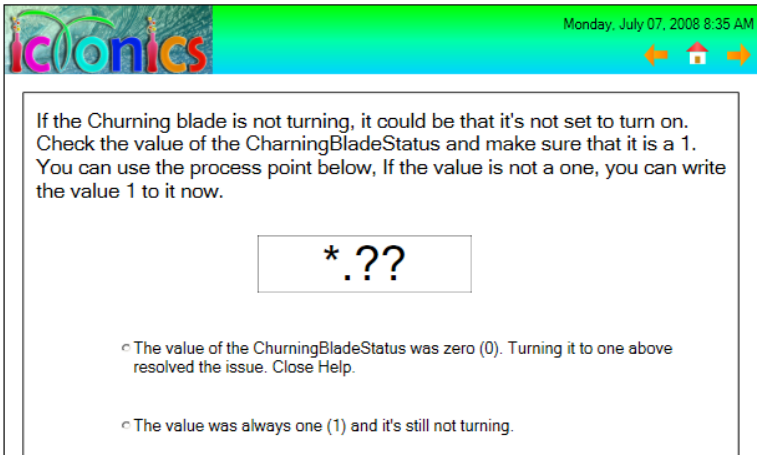


Figure L1.4 - Screen3

16. Again, the first radio button should close the window and the second one should write a value of 4 to the simulation variable.

17. Repeat step 11, and name the group Screen4. Make Screen4 look similar to Figure L1.5. This will be the last screen in the sequence.

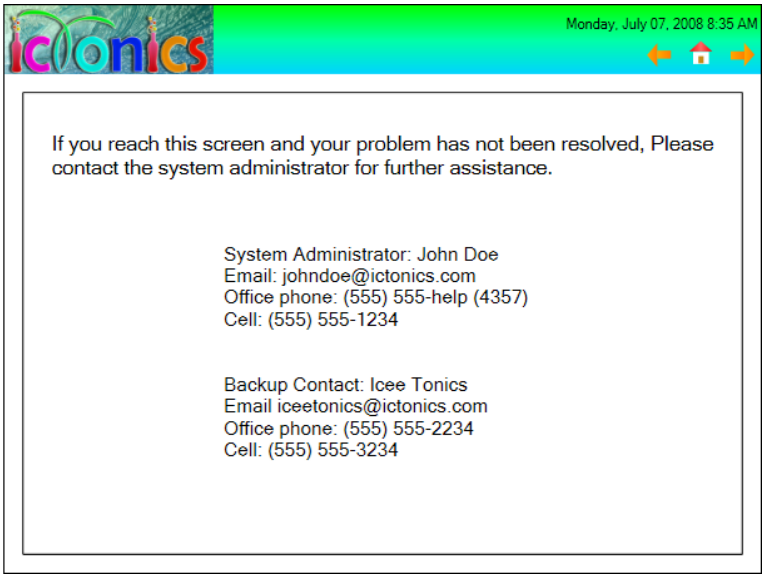


Figure L1.5 - Screen4

18. Repeat step 11, and name the group Screen5 and make it look like Figure L1.6.

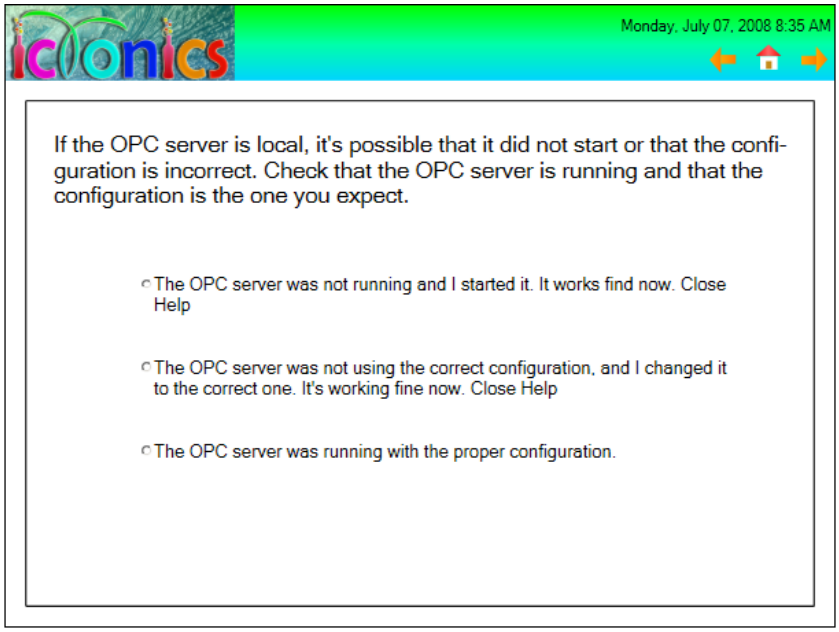


Figure L1.6 - Screen5

19. The first two radio buttons should close the window and the last one should write a value of 4 to the simulated variable.

20. Repeat step 11, and name the group Screen6 and make it look like Figure L1.7.

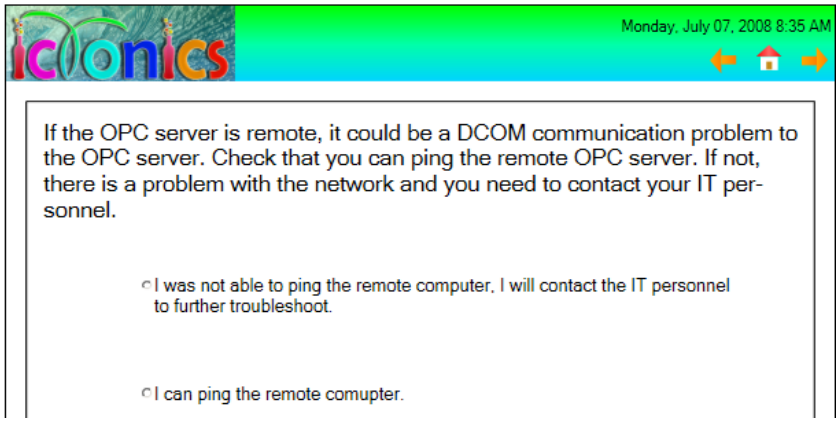


Figure L1.7 - Screen6

21. Now that we have all the screens that we need, select all of them from the explorer and click the State button in the Selectors section of the Dynamics tab of the ribbon.

22. In the DataSource field of the dynamic, select the local simulation variable that you have been using.

23. Click in the States field and click the ellipse button to bring up the Edit Selector States.

24. For each state, change the ActiveWhen to data==low. The members of the State selector should be the same order as you had them in the Explorer. Therefore, the last member is the first screen we created.

25. With that selected, (you can preview the item to the bottom left hand corner), set the low value to 0.

26. Scroll through each of the screens and assign the proper value to show the screen for the data source. When you are done configuring, click OK and exit the Edit Selector States.

27. Save your display and go into runtime to test it. If it looks OK to you, you can try launching it from the icTonics US Facility display and make sure that it works out OK.

Animations and Runtime Menu Review *R1*

1.1 Objectives Review

After completing this chapter, you should know how to:

- Configure advanced 2D animation properties
- Modify runtime settings

1.2 Summary

GraphWorX64 is a human-machine interface (HMI) software package for process control where you can create powerful, compelling, and meaningful animated graphics that display real-time data from any OPC compliant data sources. There are many dynamics available in GraphWorX64 the help you build the necessary animations.

Each dynamic available contains many advanced properties, which you will only see when developing the display in the Advanced Application Mode. Some of these properties include StartDelay, SkipInitialDuration, TargetPropertyIndex, and FlashRate. Being familiar with the advanced dynamic properties can greatly enhance the control of your dynamic in runtime.

There are also two advanced dynamics: the State and Range Selectors. Each of these dynamics allows you to group different objects and show it. The State Selector allows you to show the different objects or frames based on the value of a data source. The Range Selector on the other hand allows you plan the group of different objects in sequence within a configurable period.

Once you have configured your display with different dynamics, you can still control in runtime. Though the options are limited, but you can still easily perform zoom controls, change language, and do many other actions while in runtime. The Runtime Menu options are a great tool in getting your display to present in the format desired.

1.3 Terms

Analog Animation	Color Dynamic	Discrete Animation
Dynamics	Hide Dynamic	Location Dynamic
Range Selector	Rotation Dynamic	Size Dynamic
State Selectors		

1.4 Test Your Knowledge

In this chapter, we discussed advanced dynamic properties and some runtime menu items. Before continuing to the next chapter, make sure you can answer the questions below.

1. True or False: A dynamic in analog animation mode allows users to control how the object animates based on the value of the data source.

2. Which of the following can you do in GraphWorX64 runtime mode?

   a. Change the language of the display
   b. See a diagnostic window showing all the errors and bad quality data in the current display
   c. Print the current display
   d. All of the above
   e. Only A and B
   f. Only B and C
   g. Only A and C
   h. None of the above

3. Which property allows you to set the amount of time to run a dynamic in discrete animation mode for one cycle?

   a. Period
   b. Duration
   c. RunningPeriod
   d. CycleTime

4. Which of the following CANNOT be made into a Selector Dynamic?

   a. A grouped object such as a traffic light
   b. A group of circles
   c. A group of rectangles
   d. A group of lines

5. Which of the following are Selector Dynamics?

    a. State Selector
    b. Random Selector
    c. Range Selector
    d. All of the above
    e. Only A and B
    f. Only B and C
    g. Only A and C
    h. None of the above

Smart Symbols *2*

2.1 Objectives

After completing this chapter, you should know:

- What Smart Symbols are
- How to create a Smart Symbol
- How to add a Smart Symbol into the symbols library
- How to convert existing Symbols into a Smart Symbols
- How to change smart properties

2.2 Introductions

Smart Symbols are an enhanced version of object groups. They greatly simplify the configuration and save engineering time by allowing a configuration of properties of inner objects without the need to drill-into the symbol. You can simply configure exposed smart properties that are tied to objects within the Smart Symbol to make the whole thing work.

You can add a smart symbol from the Symbol Library to your design surface. Changing the value of a property of a Smart Symbol (called a smart property) automatically changes the value(s) of the associated properties of the descendant objects.

A typical object suitable for the conversion into a Smart Symbol is the gauge shown in Figure 2.1. When such a symbol is added into a display, it is often necessary to change the data source, which drives the rotation animation bound to the needle as well as controlling the scale on the gauge with its low and high limits.

The changes in a normal object group require you to drill-into the group using the Edit Group function or ungroup the symbol, and modify all the properties, one by one.

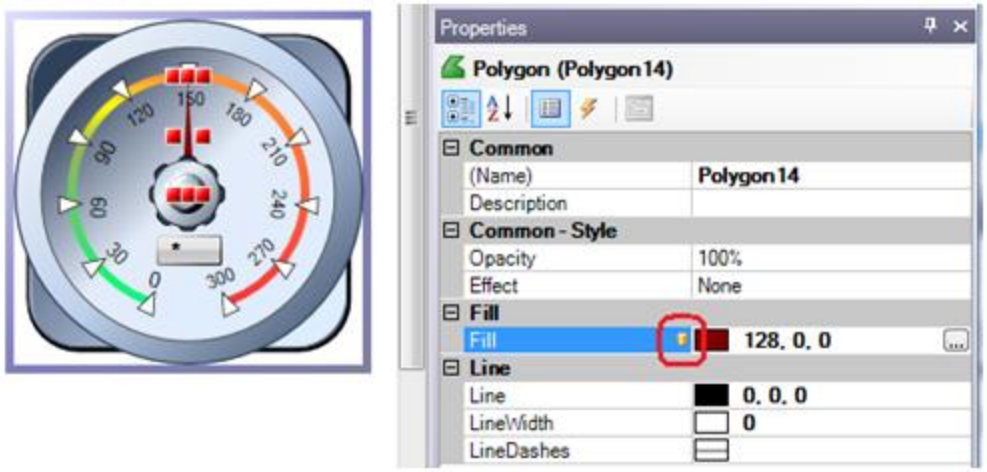


Figure 2.1 - A Typical Object Suitable to Convert into a Smart Symbol

If such a group is converted into a Smart Symbol, properties like the data source, needle color, and ranges can be exposed as Smart Properties and then configured in the properties of the top level group object.

Here is a quick list of some of the features associated with Smart Symbols:

- Smart Symbols are groupings of graphical objects. They are created from object groups using a wizard, which simplifies the binding of smart properties to properties of the descendant objects.
- Similarly, to object groups, Smart Symbols can be stored in the Symbol Library and reused later by dragging to the design surface; or copied and pasted from another location.
- Smart Symbols appear as a special kind of object in the object Explorer. They contain Smart Properties which are bound to properties of inner objects and which allow to configure the Smart Symbol.
- Smart Properties appear in the Property Pane in a section called Smart Properties.
- Smart Properties of Smart Symbols can be added, edited, and removed at anytime.

2.3 Smart Symbols in the Symbols Library

To help you understand the power of a Smart Symbol and to give you some examples, we will start our discussion with Smart Symbols by using the pre-made symbols in our Symbols Library.

You can take any symbols within the Gauges\circular gauges library, the Knobs\knobs library, or the Miscellaneous\Sliders library and apply it to your display. Changing the smart properties DataSource field will allow you to use your own data source in animating these symbols.

These symbols can also serve as an example if you wish to create your own. In the example we will present in this chapter, we will use the Gauges → circular gauges → gauge09 symbol for the most part. Feel free to browse and scrutinize the other symbols as we go through the chapter.

2.3.1 Gauge09

Let us first understand the different parts of a Smart Symbol and what makes a symbol "smart". Figure 2.2 illustrates the Gauge09 symbol from our symbols library. The symbol looks no different compared to any regular symbol at a glance. However, if you take a closer look at it, you will find some special properties about the symbol.

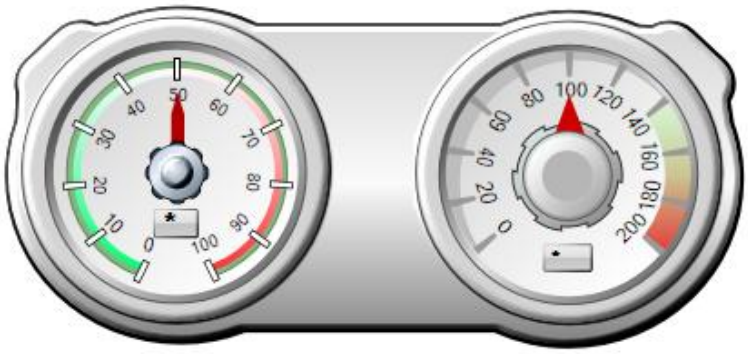


Figure 2.2 - Gauge09

First, if you look in the Explorer, you will notice that a Smart Symbol uses a different icon from regular symbols or groups. Instead of having the two rectangle icon for a group of objects, it uses a similar icon, but with an extra hand in it. Furthermore, the default name for the symbol is Smart Symbol to help indicate the type of object. Of course, you can change the name of the Smart Symbol to whatever else you desire, but the icon stays the same.

Let us now consider the properties of a Smart Symbol. A Smart Symbol has properties similar to a group with one extra section called "Smart Properties" as shown in Figure 2.3.

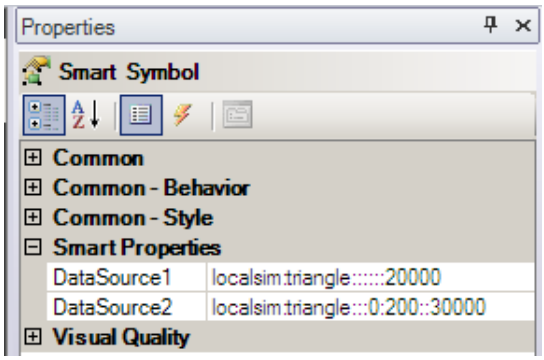


Figure 2.3 - Properties of a Smart Symbol

For now, these properties are just ones that you can set in a Smart Symbol that in turn sets the properties of objects within the symbol, allowing faster development. We will discuss Smart Properties in detail in a later section.

From the Explorer, you can still expand the Smart Symbol as if it's a regular group object and work with the individual child objects in the symbol. Notice the DataSource field of the ProcessPoint in the Smart Symbol shown in Figure 2.4 is grayed out and to the right of the entered data source, there is a small yellow cylinder icon. This indicates that DataSource of the process point is exposed as a smart property. And indeed, if we look at the Smart Properties, we see such a data source.

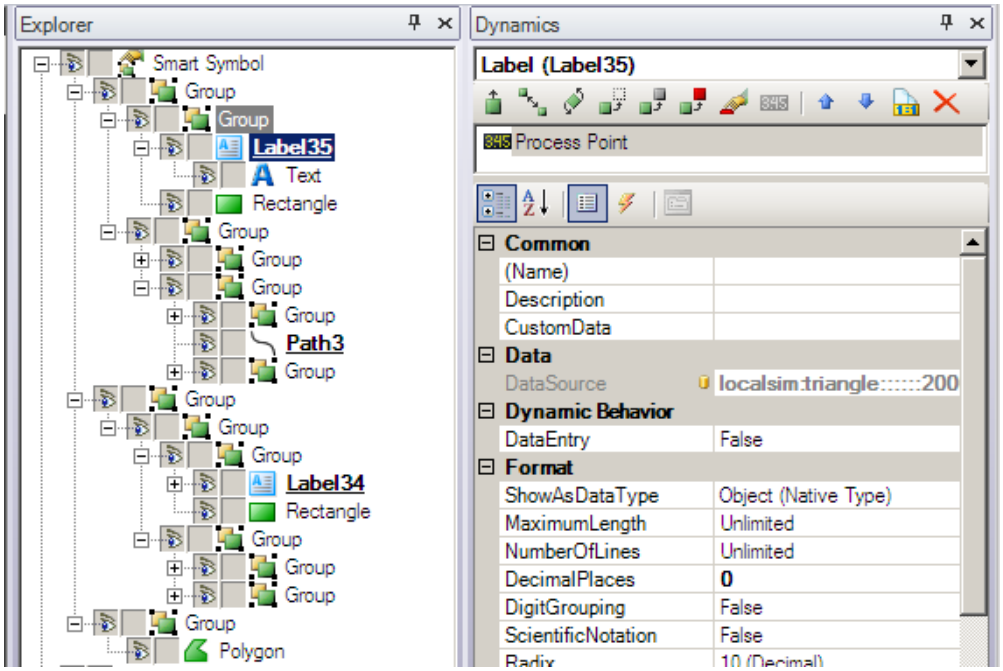


Figure 2.4 - Smart Symbol Properties

2.4 Creating a Smart Symbol

Now that we have an understanding of Smart Symbols, let us see what is required to create a Smart Symbol of our own.

In an HMI visualization product such as GraphWorX64, one of the most common property types that need to be configure is the “data source”. From process points to dynamics, all actions that we simulate on a display require some sort of a data source to control the actions.

The data source is an identifier that specifies where to get the real-time data value that drives the animations (often the data source is an OPC DA tag or OPC UA identifier that is used to get real-time data values from an OPC server). Because the data source is such a commonly used property, it is desirable to provide an automated mechanism for exposing all unique data sources used inside a group of objects as properties of a Smart Symbol. This automated smart binding can significantly reduce the amount of time needed to construct similar objects that use different data sources.

You can easily create a Smart Symbol with a few clicks and configurations. First, create the shapes you wish to include in your smart symbol, and then use the Group command to place them into a group.

Once you have a grouped objects, right-click on the shape and select **Convert to Smart Symbol** as shown in Figure 2.5. Alternatively, you can click the Convert Symbol button in the Smart Symbols section on the Dynamics tab of the ribbon.

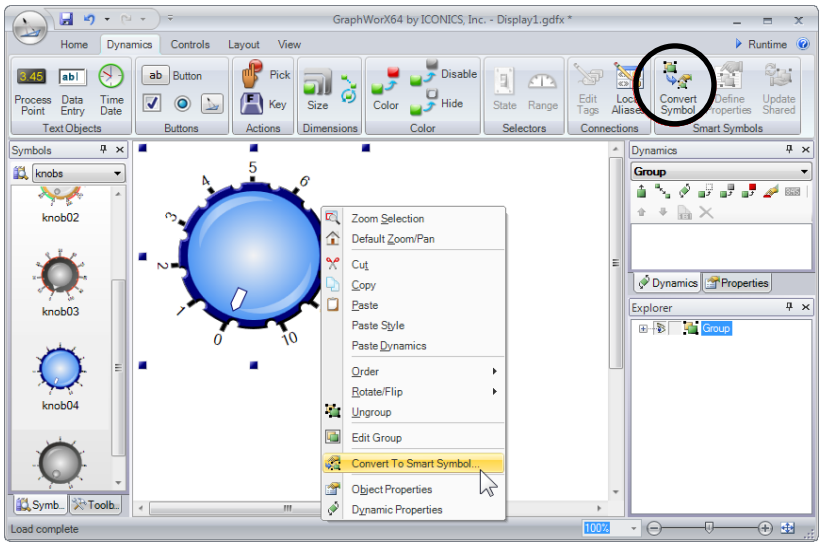


Figure 2.5 - Right-click Context Menu to Convert to a Smart Symbol

Once you choose to convert a group to a Smart Symbol, the Automatic Smart Binding Options dialog appears. There are two options in the dialog: Expose data sources as Smart Properties and Optimize hierarchical data sources as shown in Figure 2.6.

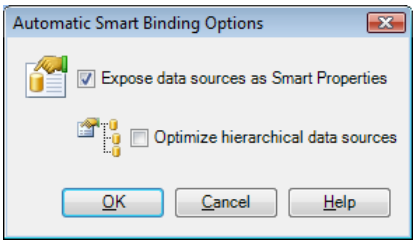


Figure 2.6 - Automatic Smart Binding Options

The **Expose data sources as Smart Properties** option is checked by default, which saves the data source properties of the smart symbol as Smart Properties so they can be changed with each use of the smart symbol; all properties that are connected to data go into the Smart Properties section. If you uncheck this option, the data sources will be saved with the smart symbol and cannot be changed when the smart symbol is reused unless you drill into the particular option with the data source attached. This is only useful when you want to reuse a smart symbol whose data sources are always the same.

The **Optimize hierarchical data sources** option is used for hierarchical data that was associated with OPC data sources to perform automatic complex data binding. When you select this option, all descendant objects of the group are examined for existing data source connections. All found data sources are analyzed to determine common hierarchical roots. Then a new Smart Property is added for each unique hierarchical root, and the descendant data source properties are automatically bound to that Smart Property by combining the Smart Property value (the common hierarchical root) with the remaining unique portion of the data source. For example, if you have the following two tags:

- opc.tcp://khu-svr2008:4840\\[KEPServerEx]icTonics/Tanks/USA/Foxboro/Tank1.TankLevel
- opc.tcp://khu-svr2008:4840\\[KEPServerEx]icTonics/Tanks/USA/Foxboro/Tank1.Producing

Instead of exposing each tag as a smart property, it makes one smart property that point to opc.tcp://khu-svr2008:4840\\[KEPServerEx]icTonics/Tanks/USA/Foxboro/Tank1 since that's the common root of both tags.

Although this is a very useful functionality in reducing redundant data sources, this functionality is only available for OPC UA tags.

Once you click OK to this dialog, the Edit Smart Properties dialog will appear as shown in Figure 2.7. In this dialog, you can **add and define** any smart properties that you want to expose in configuration mode, paying special attention to the PropertyType and DefaultValue. Depending on the PropertyType you pick, the DefaultValue field may contain different available data.

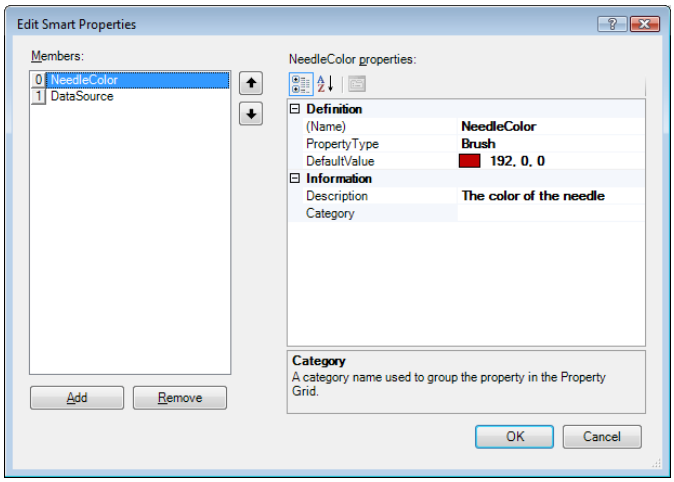


Figure 2.7 - Edit Smart Properties

Notice that if you had checked the Expose data sources as Smart Properties option in the Automatic Smart Binding Options dialog, the data sources within the group object is already listed as a smart property. All you need to do is edit the properties of the individual smart property to your liking (or you can leave it as default if you wish).

To add other smart properties, you can simply click the Add button. You will then get a new property in the Members section. The right-hand pane will give you the definitions of this property. Fill out specific information for this property. You can add as many smart properties as you wish.

Once you are done editing or adding your smart properties, you can click OK to finish creating the Smart Symbol.

2.4.1 Reusing a Smart Symbol

Once you create your symbol, it's easy to reuse it as often as you need. You can simply copy and paste it on the display or try using the shortcut keys Ctrl+D to duplicate the object. Once you have a second copy of the object, you can change the smart properties accordingly to show you the data you want to see.

The above methods are simple and a fast way to duplicate a Smart Symbol in the same display, but what if you need to use it on other displays or save it for another project later? One way is to save the display itself and each time you need to use the symbol, open the display with the Smart Symbol and copy it from the original display to the destination one. This way works, but is far from the easiest way to get the object into a display.

The easier way would be to insert the Smart Symbol into the Symbols Library so that it is always there when you need to use it. To add your Smart Symbol into the symbols library, first, choose the category into which you want to add the symbol. Then you can select the symbol, copy it and paste it into the category. Alternatively, you can hold down the control key, then select the symbol, and drag it into the category. Note that this method only works for 2D symbols; but the copy and paste method always works.

Once you add your Smart Symbol into the symbols library, you can always access it whenever you open a GraphWorX64 display.

2.5 Editing a Smart Symbol

Once you create a Smart Symbol and save it, you may want to change it at times; after all, this is just a base object created for simplicity when creating displays that reuses common parts.

You can edit a Smart Symbol the same way that you edit any grouped objects. In the Explorer, you can drill into the different parts of the symbol and then insert additional objects into the symbol. You can also double-click on the symbol in the display to drill into it. The Explorer just provides a clearer view of the Smart Symbol's hierarchy of objects.

One of the most common things to edit in a Smart Symbol is probably the Smart Properties. By now, you probably have figured out that a Smart Property is used to set one or more properties of child/descendant objects contained in the Smart Symbol. A child object is not limited to "visual" objects; a child object can also be a non-visual object such as an animation definition (a.k.a. a Dynamic).

In the previous section, we have discussed how we can add Smart Properties into a Smart Symbol in the process of creating the symbol. Let us now look at how we can add a smart property after creating a Smart Symbol.

2.5.1 Adding Smart Properties

Once you edit a Smart Symbol, chances are you would like to make or bind certain properties Smart Properties. Well will examine how we bind a property in the next section, for now, let us discuss how to add Smart Properties after you have made a group of objects into a Smart Symbol.

Let us take a quick detour and first understand what makes up a smart property. From the earlier section, you saw the Edit Smart Properties dialog (shown in Figure 2.7) and from there, you probably gathered that a smart property definition consists of the property name, data type, default value, description, and category.

Consider our gauge09 Smart Symbol. It contains two needles for the two gauges. We can expose both needles' color as a smart property, allowing us to change the color of both needles at the same time at the top level.

To do this, we will first select one of the needles. For the sake of discussion, we will choose the one on the left-hand side (which is Path3). Since we want to change its color, in the properties of the object, look for the Fill property. Once you find it, right-click on the field and select "Add to Smart Properties".

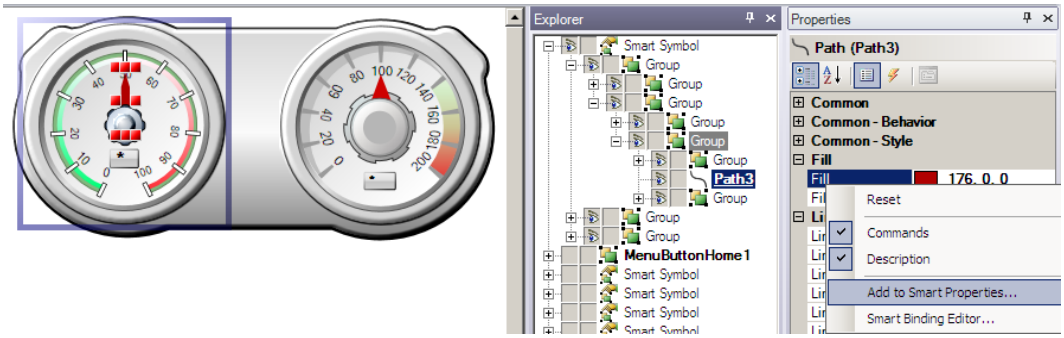


Figure 2.8 - Properties of Gauge09

Once you select the option, you will get the Add Smart Property dialog as shown in Figure 2.9. It will give you a default name based on the type of property you want to make into a Smart Property. If you don't like the default name, you can type in your own.

Figure 2.9 - Add Smart Property Dialog

Once you are done with giving the property a name, click the "OK" button. This action adds the property, along with its current value (as the default) to the Smart Properties list. A two click action therefore replaces opening the Editor, creating a new property, determining its PropertyType, setting a default value, and such.

Now, you can go back to the top level of the Smart symbol and see the additional Smart Property we just introduced. You can click on the property field and change the color. You will notice that once you pick a different color, the left needle's color changes.

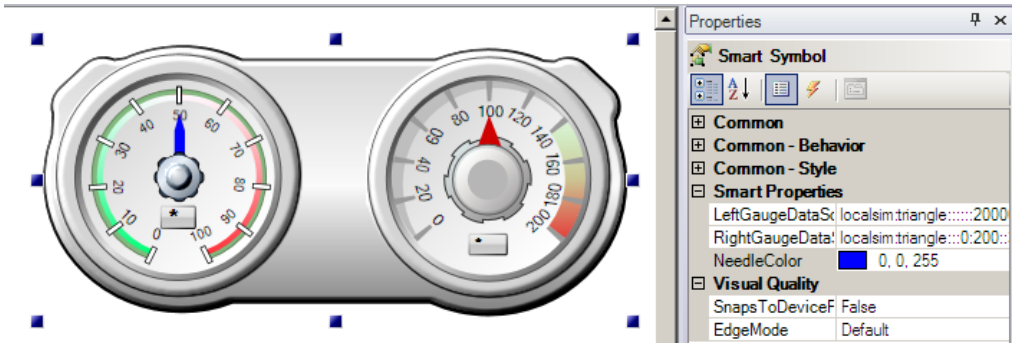


Figure 2.10 - Color Changed on the Left Needle

If you wish, you can right-click on the new Smart Properties property and select the "Edit Smart Property…" option to view and edit the properties of the Smart Property.

2.5.2 Smart Property Bindings

After you have inserted objects or dynamics into a Smart Symbol, you can associate the already exposed smart properties to these objects. This way, you can configure the property of the particular object when you change it on the parent level.

The process of associating a regular property to a smart property is called **Property Binding**. You can do this by right-clicking on the property in which you want to add as a smart property and select "Smart Binding Editor…" in the right-click context menu. Keeping with our previous example, we will select guage09's right needle (which is Polygon21), right-click in its Fill property and select the binding option as shown in Figure 2.9.

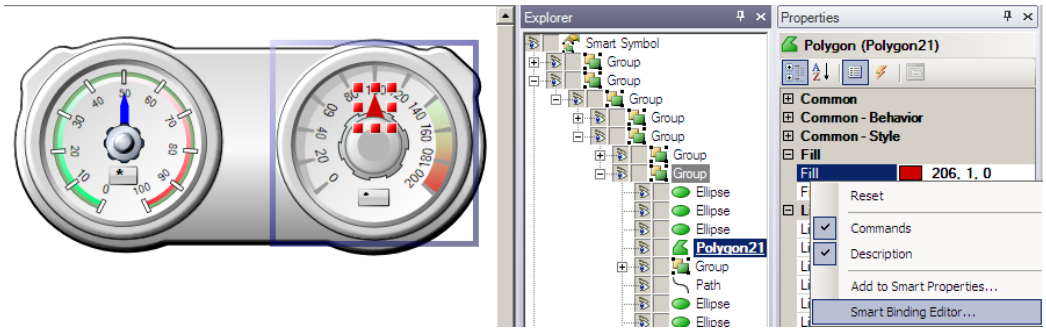


Figure 2.11 - Right-click Context Menu to Bind Smart Properties

Once you click on this link, the Edit Property Binding dialog will come up and you can select any of the smart properties available on the left-hand pane and click the “Add” button to add it. If there are additional logics that you want to insert, you can change it in the Binding Logic section on the right-hand pane.

Since all we want to do here is to associate this property with our needle color Smart Property we created before, we will only select the property on the left-hand side, click the "Add" button and then click on "OK" to exit out of the dialog.

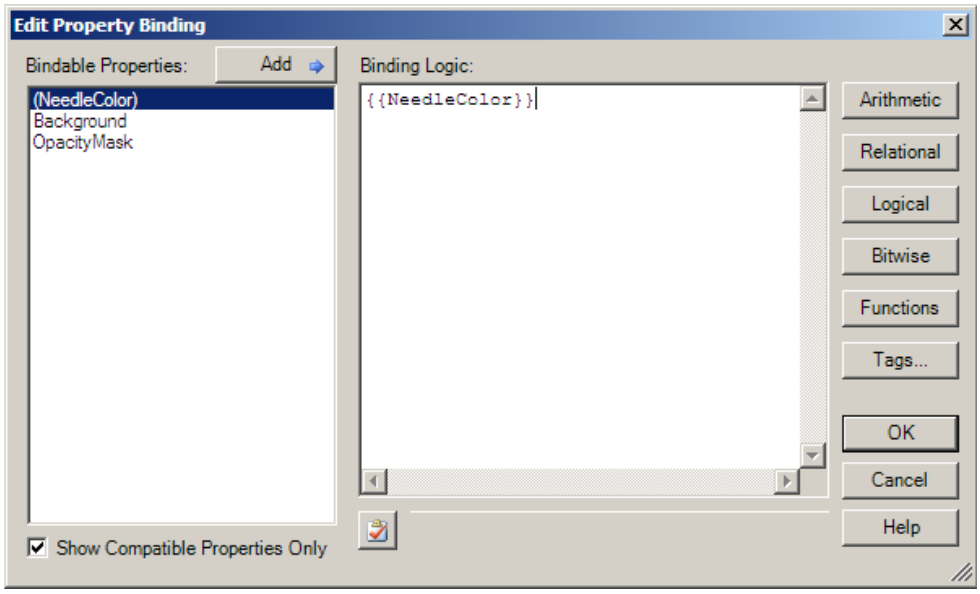


Figure 2.12 - Edit Property Binding Dialog

When you click on "OK", you will be returned to the GraphWorX64 display. The property that you select to bind to a Smart Property now has a yellow cylinder icon next to it and the properties field is grayed out showing you the Smart Property value. Also, the default color set for the property takes effect and the needle changes to that color as well.

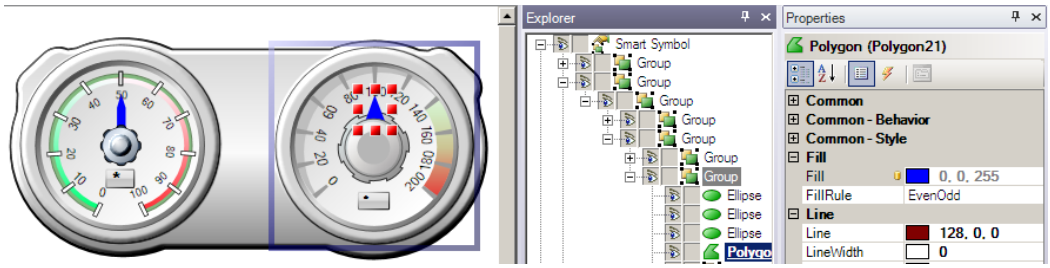


Figure 2.13 - Right Needle Color Change

Now, if you go back to the top level of the Smart Symbol and change the value of the smart property, all children properties that are associated to the smart property will change with it. In our case, setting the color of the NeedleColor property in our Smart Symbol to another one, changes the color of both needles at the same time as shown in Figure 2.14.

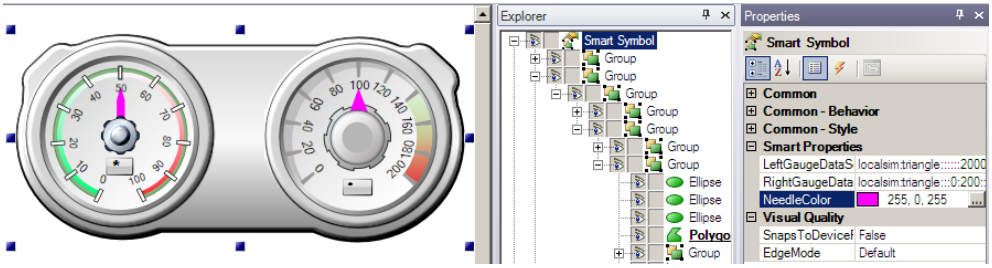


Figure 2.14 - Both Needle Color Changed by Changing One Property

Lab: Smart Symbols 2

2.1 Lab Overview

IcTonics loves what we have done with the bottling line. There is just one thing, they would like the line to be automatic with a switch as oppose to the different radio buttons that we were using. Also, since they have different lines running at the same time, they would like to make it so that their engineers can use some symbols that we build later on, change the tags associate in a few clicks and start a new bottling line.

We will try to accomplish this task in this chapter with smart symbols and simple expressions.

Estimated time to complete this lab: 20 minutes

2.1.1 Task Checklist

In this lab, you will complete the following tasks:

- Create a smart switch to control the bottling line
- Create a smart bottling line

2.1.2 Support Files

You will need the following files to complete this lab:

- icTonics US Facility.gdfx

2.2 Familiarizing with Smart Symbols

We have already created a Smart Symbol during the 3D course; we will take some time to understand it now.

1. The Smart Symbol we have created in the 3D course is called Smart Tank and you should find it in your GENESIS64TrainingSymbols library.

2. There are three smart properties associated with this tank: a local simulation variable that controls the locations dynamic of the top of the tank, the churning blade status tag, and the tank level tag.

3. If you don't remember how each of the tags is used in the display, you can drill into the symbol and look at the different dynamics to find out how it works.

4. The tank was our first introduction to Smart Symbols. In the next section, we will expand on this topic.

2.3 Smart Switch

In this section, we will create a switch, which controls the bottling processes. When the process begins, we will start the churning blade in the tank, as well as begin the bottling conveyor line. Since we have different production lines, we will also make this a Smart Symbol so that we can associate different data sources to it later.

1. Open the icTonics US Facility.gdfx display if you closed it from the previous section.

2. Since there are no preconfigured switches available, we will quickly make one our self.

3. The switch we will build contains two rectangles for the plate, and a switch with two states, which contains two more rectangles and two circles. Your switch may look similar to the one shown in Figure L2.1.

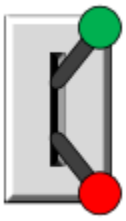


Figure L2.1 - Simple Switch

4. You should also add a hide dynamic (or you can try the State Selector) to the switch so that when it's in the On state, the Off position is hidden and vice

versa. You can add a Toggle Value Pick Action to the entire switch so that that state changes each time you click on it.

5. Once you are happy with the switch, with it selected, on the Dynamics tab of the ribbon, click the Convert Symbol button in the Smart Symbols section.

6. The Automatic Smart Binding Options dialog comes up. You can leave the selection as default and click on "OK" to continue.

7. You should have a smart property in there since you used a data source to control the show and hide of the state as well as the pick action to switch the state.

8. Let's rename this Smart Property to "OnOffTag". You can add an option description if you wish. Your Edit Smart Properties dialog should look similar to Figure L2.2.

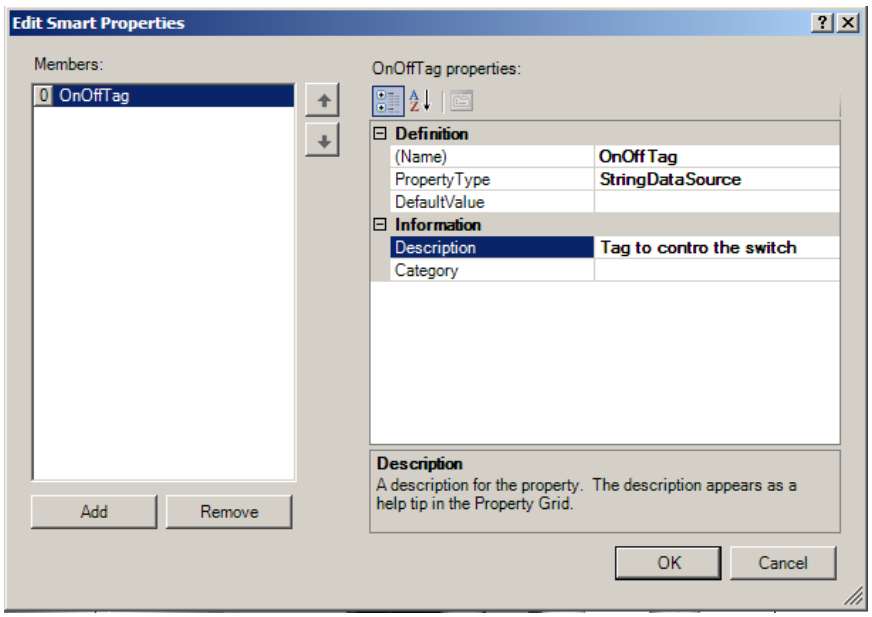


Figure L2.2 - Edit Smart Properties Dialog

9. When you are done in the Edit Smart Properties dialog, click on "OK" to exit it and finish creating the Smart Symbol.

10. We can name this "SmartSwitch". Hold down the Ctrl key and select your SmartSwitch and drag it into the trainings symbols library. This way, we can reuse it later.

11. Now, you can make your entire line a smart symbol and store it in the symbols library. We will just make a couple of changes to it.

12. Select all the bottles and the conveyor and group it, then select the option to make it into a smart symbol. Uncheck the “Expose data sources as Smart Properties” option and click on "OK".

13. Since we did not automatically add any smart properties, you will not see any in the Smart Symbol’s property. However, we can add some by right-clicking in any field of the Properties section and select “Edit Smart Properties…” option.

14. In the Edit Smart Properties dialog, click the Add button to add a new property.

15. Let’s call this smart property BottlingLine and set the Property Type to StringDataSource. For the default value, let’s create a new simulation tag called “bottle”. You can leave everything as default for this simulation variable.

16. The idea here is to automate the line when we turn on the switch. To do this, we are going to control the four steps we have in the control board with just one variable. So we need to change some of the dynamics created in this display.

17. First, for each location dynamics on a bottle change it so that the data source is bind to the smart property. Then change the ActiveWhen state to data== low and set the low limit to 1. The Data section of the one of the dynamic should look like Figure 2.3.

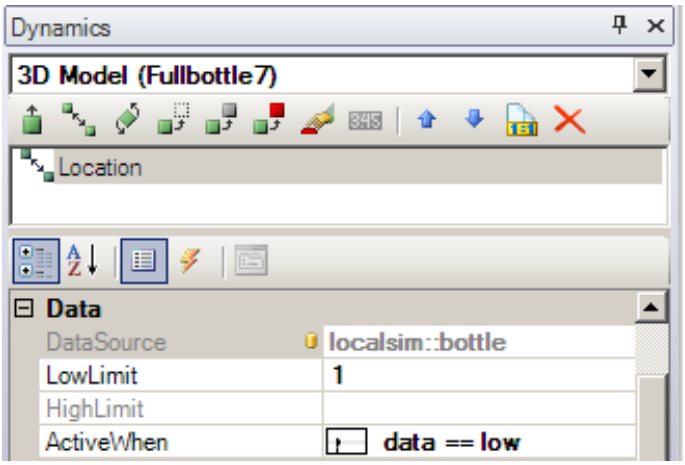


Figure 2.3 - Configuration of a Location Dynamic on a Bottle

Note: You can try to copy the dynamic to each of the objects after changing the Data Section of one location dynamic. Just be sure that if you do this, double check that the end location is still correct.

18. On the overlaid bottles (one that's fill and contains the locations dynamic and one that has the fill dynamic), change the Hide Dynamic on the bottle with the Locations dynamics so that it hides the bottle when the tag is greater than 1 and greater than or equals 3. Don't forget to bind the DataSource to the smart property.

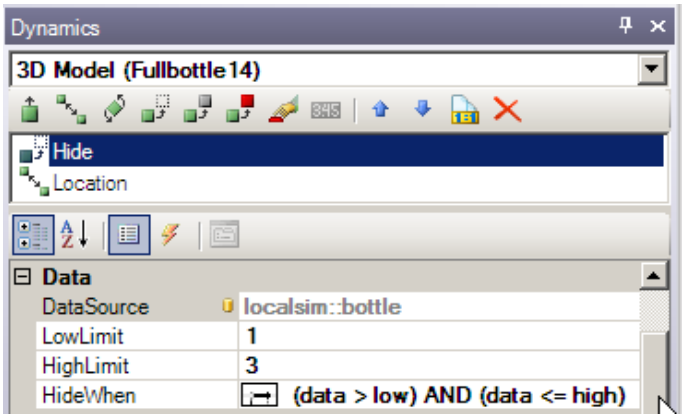


Figure 2.4 - Hide Dynamic for First Full Bottle

19. On the Hide dynamic for the bottle that has the fill dynamic, make it so that the bottle shows when the tag has a value between 1 and 4. Again, you should bind the data source to the smart property.

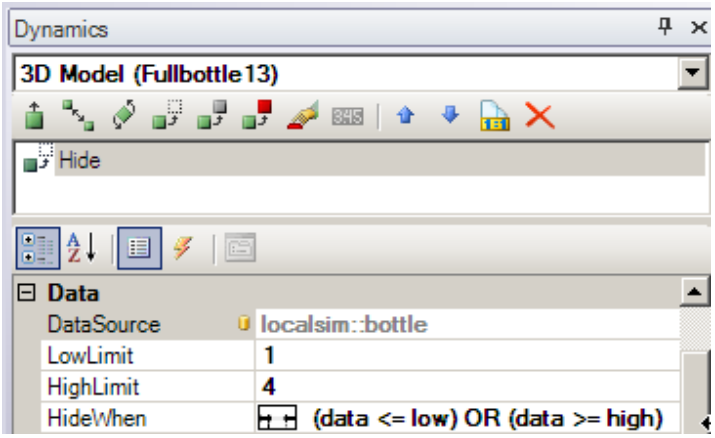


Figure 2.5 - Hide Dynamic for the Filling Bottle

20. Now drill into the filling bottle and find the fill. Change the size dynamic here so that it will start when the tag has a value of 3. You should bind the DataSource to the smart property once more.

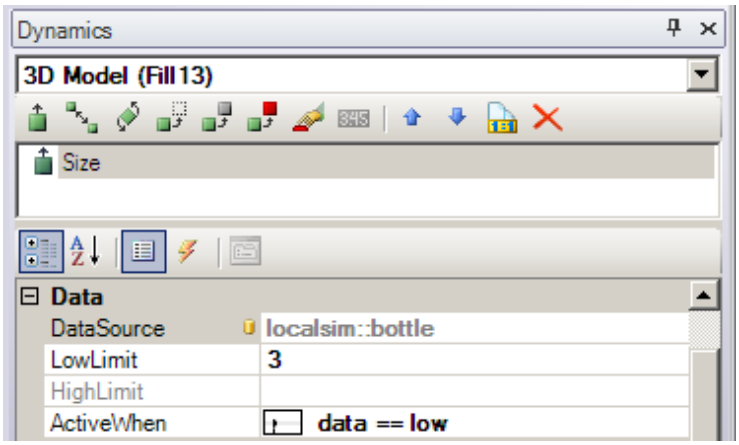


Figure 2.6 - Size Dynamic for the Fill of the Bottle

21. The last thing is the extension tube that fills the bottles. You can change it so that the tube extends between the values 2 and 3. Remember to bind the DataSource to the smart property.

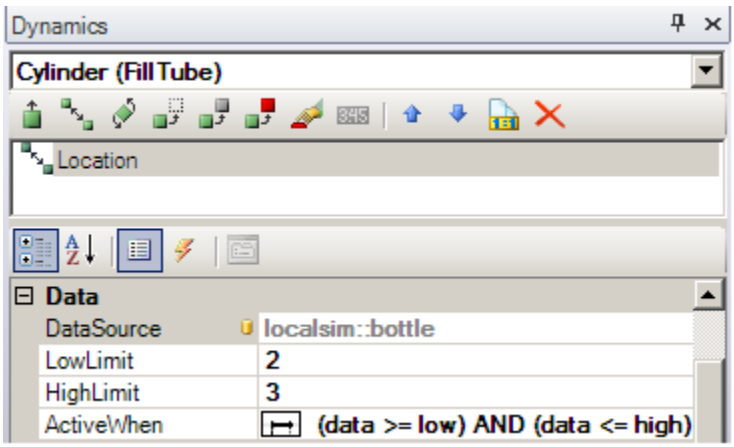


Figure 2.7 - Location Dynamics for the Extension Tube

22. By now, you probably have figured out that depending on the value of "bottles", it will control the stage of the bottling line.

23. We will now configure a checkbox that will change the state of "bottle" for us. Add a check box to the display and set the Pick Action to Write Value.

24. The data source should be the bottle variable. Set the Execution Trigger to ExecuteWhileChecked and set the WhileCheckedInterval to 2000.

25. In the WhileCheckedValue, input the expression seen in Figure 2.8.

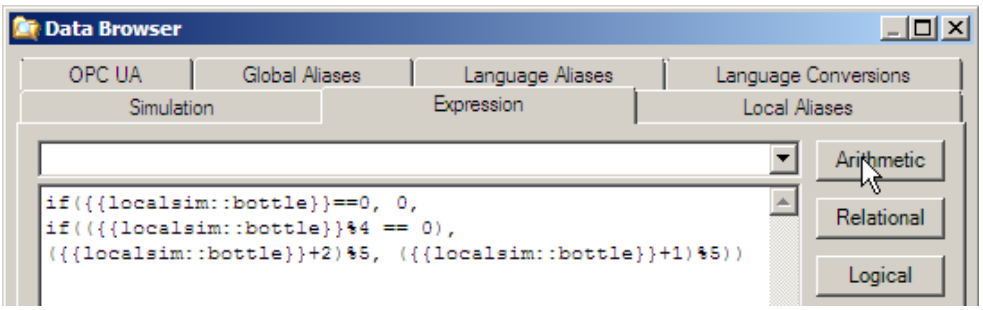


Figure 2.8 - Expression to Automate Bottling Process

26. Your Pick Action configuration should look similar to Figure 2.9.

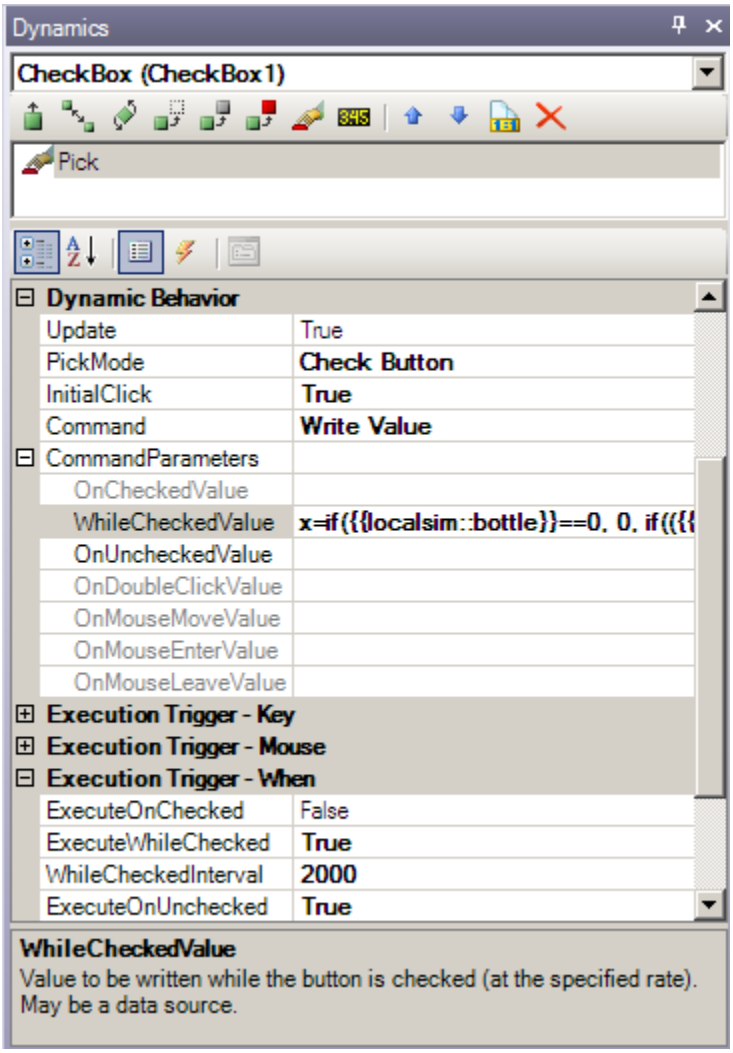


Figure 2.9 - Checkbox Pick Action

27. Now, let's also set the data source for our switch to the bottle variable.

28. You can hide the radio buttons in the control board and move the switch into that section. You can hide the checkbox as well since we don't need to interact with it in runtime.

29. Go into runtime, try flipping the switch between the on and off state and see your bottling line run.

Smart Symbols Review R2

2.1 Objectives Review

After completing this chapter, you should understand the following topics:

- What Smart Symbols are
- How to create Smart Symbols
- How to add a Smart Symbol into the Symbols Library
- How to convert existing symbols into Smart Symbols
- Smart Properties

2.2 Summary

Smart Symbols are an enhanced version of object groups. They greatly simplify the configuration and save engineering time by allowing a configuration of properties of inner objects without a need to drill-into the symbol. A smart symbol can be configured without needing to “drill-into” the symbol to make it work.

You can add a smart symbol from the Symbol Library to your design surface. Changing the value of a property of a Smart Symbol (called a smart property), automatically changes the value(s) of the associated properties of the descendant objects.

2.3 Terms

Smart Properties Smart Symbol Symbols Library

2.4 Test Your Knowledge

In this chapter, we discussed many aspects about Smart Symbols: from what they are to how to create one, edit one, and store one for later use. Before you continue to the next chapter, be sure that you can answer the following questions.

1. True or False: You can convert any grouped objects into a Smart Symbol.

2. True or False: A Smart Symbol is an ActiveX.

3. True or False: You can store Smart Symbols in the Symbols Library and reuse them later.

4. True or False: You can only edit Smart Symbols on the symbols level and not on the individual object level.

5. A Smart Symbol has the following features:

   a. You can expose data sources as smart properties automatically when creating a Smart Symbol
   b. You can easily add, edit, or remove Smart Properties from a Smart Symbol
   c. You can change a Smart Property that is linked to five other properties in the Smart Symbol at the same time.
   d. All of the above
   e. Only A and B
   f. Only A and C
   g. Only B and C
   h. None of the above

6. Which of the following CANNOT be made into a Smart Symbol directly

   a. A group of controls
   b. A group of process points
   c. A group of dynamics
   d. A single line object

Scripting Basics *3*

3.1 Objectives

After completing this chapter, you should know how to:

- Use the Script Editor in GraphWorX64
- Use GraphWorx64 global variables
- Manipulate OPC tags through Scripting
- Insert objects and change properties through Scripting
- Launch the Security Login through Scripting

3.2 Introductions

GraphWorX64 offers you the possibility to write scripts to define some custom behavior and to interact with the product. The possibilities offered by scripting are endless, but let us introduce some examples:

- You can read from and write to an OPC data source
- You can write your own logic to manipulate and perform computation over your data
- You can use scripts to connect to a remote database and pass that data to some GraphWorX64 component for visualization
- You can alter GraphWorX64 objects' properties
- You can dynamically create and control the content of a 3D View control
- You can call GENESIS64 API functions to take full control of the product

This is just a few possible examples. GraphWorX64 utilizes JScript.NET, which is a programming language in the.NET family. JScript.NET allows you to call .NET framework functions, as well as to interact with .NET code from any .NET DLL. This means you can write your own DLL and use it inside the product, interfacing to it through scripting.

Given that JScript.NET is a .NET language, being familiar with .NET programming will be most helpful for you. Apart for some syntactical differences and a few limitations, JScript.NET is no different from other well-known .NET languages such as C# and VisualBasic.NET.

Although scripting is a very useful feature, you should still remember that GraphWorX64 is by itself a very powerful product. For this reason, you should not abuse the power of scripting to perform tasks that can easily be carried out by using the built-in feature of GraphWorX64. For example, you may be tempted to write a short script to write a value to an OPC data source, but you can do this in no time by using a "Write Value" Pick Action in GraphWorX64 or simply use the Data Entry Dynamic. On the other hand, you may want to write a script to read several different OPC data source values, apply some logic on the values and execute some command only if a specific condition is met. In this case, it's perfectly fine to use a script, but you may want to consider using an expression to evaluate your condition instead. This is just a matter of personal preferences, of course. If you like to write your own code, we will not stop you from doing it, however, we strongly suggest that you first try using the built-in features whenever is possible.

You will probably find that over time, you will need to create more powerful custom projects that require scripting in GraphWorX64. For example, when a display is loaded into runtime, you may want to prompt the user with an instructional message box, or a form, that might not be available through the GraphWorX64 configurator.

We will now explore several concepts that will be useful for working with scripting along with a few examples.

3.3 Using the Script Editor

To let you create your own script, GraphWorX64 provides a Script Editor. This window is usually docked on the lower left hand side of the display, as shown in Figure 3.1. You can open and close the Script Editor by pressing the Alt + F11 shortcut command. The Script Editor, as well as other dockable windows in GraphWorX64, comes with an Auto-Hide feature. Along the top of the Script Editor, there is an Auto-Hide pin. You can click this pin so that it changes its orientation to sideways , which means it will minimize itself when you are not actively using it.

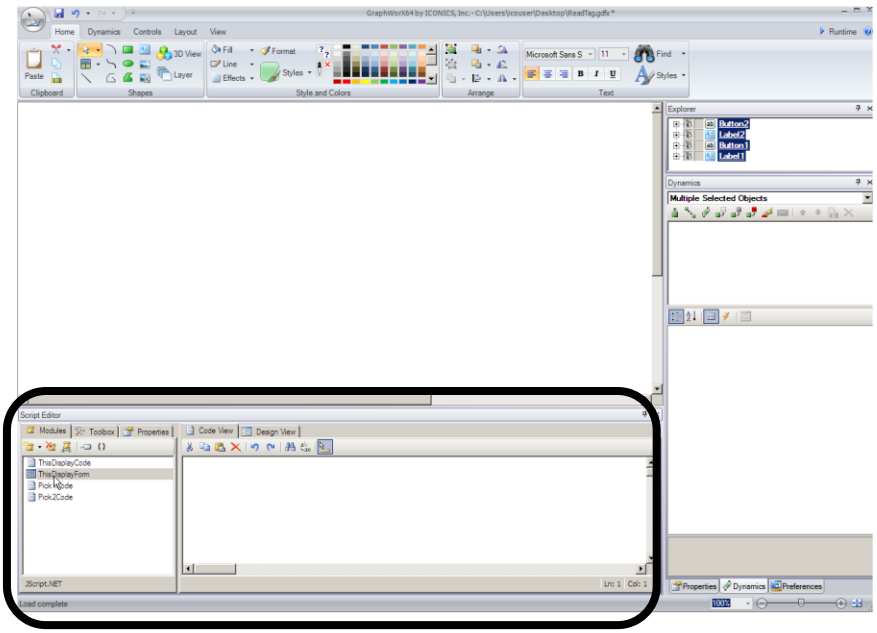


Figure 3.1 - GraphWorX64 Script Editor

There are a total of five tabs in the script editor and include:

- **Modules:** Allows you to create and modify script modules.
- **Toolbox:** Allows you to import selected Windows form control into your scripting forms.
- **Properties:** Allows you to configure properties of a form created in the script editor.
- **Code View:** Allows you to view the source code of your script.
- **Design View:** Available when a form exists in the script editor. Allows you to visually see your form and edit it without writing code.

One of the most useful features of the Script Editor is IntelliSense, which can be used when writing code in the Code View. Using IntelliSense makes writing scripts tremendously easier because as you are typing, you will see a tooltip that lets you see the available attributes for that object, as well as a description of what that attribute does. For example, you may want to show a Message Box, but forgot the correct functions to apply. Using IntelliSense, as shown in Figure 3.2, you can browse through the available objects, properties, and methods, taking advantage of the AutoComplete feature, as well as visualizing a description of the highlighted item, MessageBox in this example.

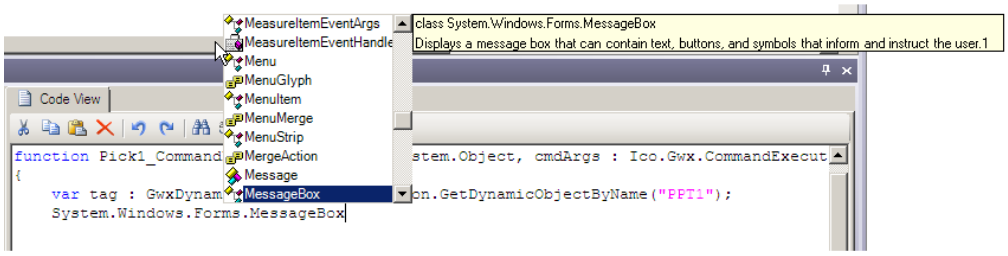


Figure 3.2 - Using IntelliSense in Code View

However, keep in mind that IntelliSense is just a tool to help make scripting easier. If it does not show you what you need to know to use a function, you should refer to the *GENESIS64 API Reference Help* located in the GENESIS64 start menu.

Once you are in the script editor, you will see that there is a toolbar in the Modules tab. The toolbar allows you to create and modify script modules. It includes the following features, as listed from left to right in Figure 3.3:

Figure 3.3 - Modules Toolbar

- **Add New Script Module:** Creates a new script module in the Modules tab and allows a name for the module to be specified from the Add Modules dialog box.
- **Delete Selected Script Module:** Removes the selected script module from the Script editor.
- **Rename Selected Script Module:** Allows you to rename the selected script module.
- **Edit References:** Opens the References dialog box, where you can edit script DLL references.
- **Edit Namespace**: Opens the Namespace dialog box, where you can create shortcuts to use with scripts.

NOTE: For more information about the features of the script editor, please refer to the Help Documentation included in GraphWorX64.

3.4 General Scripting Practices

As you become more familiar with scripting in GraphWorX64, you will come across several features that you will see time and time again. In this section, you will learn how to create a script, and when to use the more common global variables.

3.4.1 How to Create a Script

You can run a script on a wide range of events, but the most common way is to run a script from a button. In this example, you will see the steps needed to set a script to run from the click of a button in runtime mode.

Starting in configuration mode, add a button to your display and select the RunScript command for its Dynamic Behavior. From the Dynamics tab, select the Events button. This will display the possible events related to this dynamic, which in this example is just the RunScript event.

There are two ways of naming this script. The first is to double-click on the RunScript event, which creates a function by using the dynamics' name and the dynamic performed. The second way is to enter a name for the script yourself manually.

Whichever way you use to create the script, after double-clicking, or after pressing enter once you entered the name of a script, the Script Options dialog will appear, as shown in Figure 3.4. The dialog gives you the option to create a form by checking the Add a new Form check box. If you choose to create a form, it will be displayed when the script runs. You do not have to create a form if it is not necessary. To complete the process of creating a script function, simply click OK in the Script Options dialog and you will be brought to the Code View in the Script Editor.

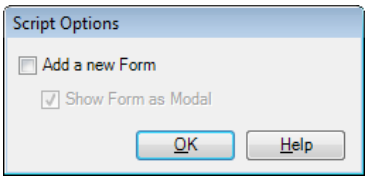


Figure 3.4 - Script Options Dialog

It makes a slight difference in creating modules depending on how you created your script and the name you gave it. As mentioned before, if you double-click on the RunScript, it creates a function with the name of the dynamic and what the dynamic performs and your function name may end up being something like Pick1_CommandExecuted. If you scrutinize the Modules list, you will notice that there is also a module called *Pick1Code*. And if you have chosen to create a form with it, there will be a form in the module tree call *Pick1Form*. However, if you simply type in the name of a script, for the sake of our example, let us call it *Script1*, you will have a function called just that, but there is no module created where the function resides. Instead, this function resides in the ThisDisplayCode module. If you had a form created with it, it would be called *ThisDisplayForm*

(for the first one, and then *ThisDisplayForm2*, for the second one and so on). If you would like to type in your own module name and the function name, then you must type in both when you create the script, separating it by an underscore (_). For example, if you want a module named *myModule* and the function to be *myFunction*, then when you type in the name of the script, you should type in *myModule_myFunction*. Anything in front of the underscore will be treated as the module name. Appended to this name is the word Code and therefore, the entire name of the module will end up being *myModuleCode*.

In this example, the script name was automatically generated and the Add a new Form checkbox was not checked. From here, you will be brought to the Script Editor in Code View where an empty function has been created for you:

```
function Pick1_CommandExecuted(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
}
```

At this point, you can now write your code that will run when this script executes. For example, you can enter the code shown below in the function body, to create a Message Box that displays “Hello World!”

```
function Pick1_CommandExecuted(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      MessageBox.Show("Hello World!");
}
```

Once you complete the code, you can simply go into runtime mode and click the button to see the Message Box as show in Figure 3.5.

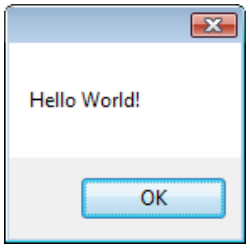


Figure 3.5 - Hello World Message Box

3.4.2 GraphWorX64 Global Variables

GraphWorX64 exposes a few different global variables that you can use to access different elements of the application quickly. These variables, for example, allow you to interact with the currently loaded display, to retrieve objects, symbols, and dynamics to work with them.

You can consider these variables as the "entry points" to the GraphWorX64 object hierarchy. Most of your scripts will use these variables, and for this reason, it is very important to understand them, and to know when to use which in your scripts.

- **ThisConfiguration** – Related functions to manipulating objects such as process points, rectangles, etc. If you are familiar with scripting in GraphWorX32, then this would be most comparable to using ThisDisplay.
- **ThisDocument** – Related to all the runtime functionality including the options found on the Properties tab in configuration mode.
- **ThisWindow** – Related to the view area of the display, such as the dimensions of the viewing area, or converting point coordinates.

To understand how these variables should be used inside a script better, consider the following example. We will slightly change the "Hello World!" example we built earlier. Suppose we need to interact with an object on the display to read some of its properties. In this case, we will try to read the button's caption since it's the only object we have on the display.

All you need to do to have this example working is to change the body of the function we worked with earlier, with these two lines, which we will analyze:

```
function Pick1_CommandExecuted(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      var b : GwxButton =
      ThisConfiguration.GetObjectByName("Button1");
      MessageBox.Show(b.Text);
}
```

The first line of this simple script contains several new interesting concepts, other than using a global variable. You can see how to declare a variable, and how to specify its type. You can also see that the type of a GraphWorX64 Button is GwxButton. Finally, you can see how to use the `ThisConfiguration` variable to retrieve an object on the display by mean of the `GetObjectByName` function. Please note that the parameter of this function is the button's name. If you changed the name of your button, you also need to change this string to match with your chosen name.

The second line is no different from our previous example, except for the message itself. Instead of using a plain string "Hello World!" we use a property from our button: its Text property.

When you go into Runtime Mode and you click on your button, you should see something similar to Figure 3.6.

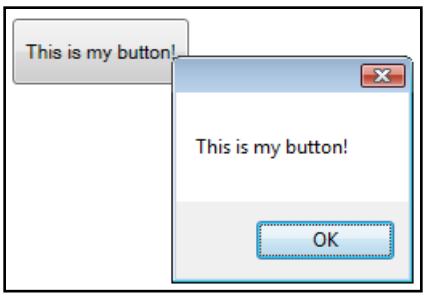


Figure 3.6 - Reading the Text Property from a Button on the Display

3.5 Scripting Examples

As we have said in the Introduction, your possibilities to create custom behavior and to take control of the product are endless when using Scripting, the same way you have unlimited possibilities when writing your own software using any programming language. For this reason, a complete coverage of Scripting is not possible, and the only way to master this technology is through experience and by studying existing examples. Therefore, this section introduces you to some other basic examples, which you may be considered as a building block that you can combine together to create more complex scripts for use in your application.

3.5.1 Reading from an OPC Tag

One of the most important features of GENESIS64 is its capability to interface with many different data sources. This data is used to animate your graphics elements and to perform a number of other tasks. For this reason, it is quite important to be able to access this wealth of information also through scripting, so that you can create your own logic to process and consume that data.

There are several ways to read an OPC Tag through scripting, and in this section, you will learn one of them. In this example, we will trigger our script by clicking on a button (as in the previous examples). The script will read a couple of data source values and will present them in a message box dialog.

First, we need to prepare our display. Place a Process Point and define localsim:sine as its data source. Then, give this dynamic a name, such as “PPT1”. Second, we need to add a Data Entry object (which is like an editable Process Point). Define a new simulated variable and name this object as “PPT2”. Your dynamic object configurations should look similar to the one in Figure 3.8.

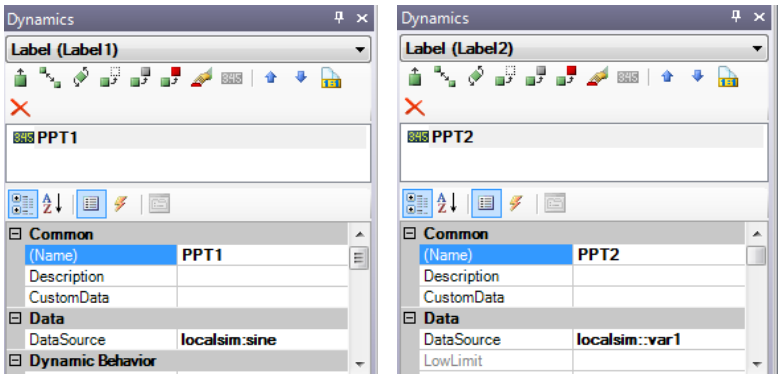


Figure 3.7 – PPT1 (Left) and PPT2 (Right) Configurations

Now that we have created the objects we need to work with, we can proceed with creating a button to run a script. You should be familiar with the procedure to create a button and let the system generate a new script function for you. If you feel uncomfortable in doing this, please review the previous section.

We can now start writing the script function, which will read and show the data source values. This is not too different from the previous example where we have read the button's text. Below is the necessary code:

```
function Pick1_CommandExecuted(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      //declare variables
      var ppt1 : GwxDynamic =
      ThisConfiguration.GetDynamicObjectByName("PPT1");
      var ppt2 : GwxDynamic =
      ThisConfiguration.GetDynamicObjectByName("PPT2");

      //read values and output into a message box
      MessageBox.Show(String.Format("The first Process
      Point value is: {0}\n" + "The second Process Point
      value is: {1}",

      ppt1.GetValueOfPrimaryDataSource(),

      ppt2.GetValueOfPrimaryDataSource()));
}
```

The first section is very similar to what we used to get a reference to the button earlier. In this case, we use a different function because we want to get a dynamic object, instead of a regular object. Once we have retrieved the two process point objects, we can show a message with their values. The function that gets a process point value is called `GetValueOfPrimaryDataSource`.

The result of running such a script is shown in Figure 3.8.

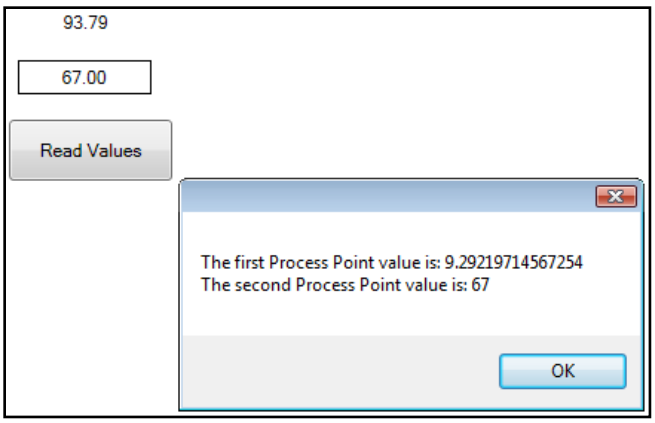


Figure 3.8 - Reading Two Process Point Values

NOTE: The first process point value is actually different from what is shown in the message box. This is because the GraphWorX64 display is still animating behind the message box, thus the process point value is continuously changing.

3.5.2 Writing to an OPC Tag

In this section, we will learn how to write a value to an OPC tag. We can reuse the previous example, and simply enhance it, by adding a new button running a different script to write some value to a data source.

The process point where we want to write the value to is the PPT2 Process Point from the previous example. To make it more interesting, the value we are going to write will be the value read from PPT1. For this reason, we do not need to do anything particular to change our display other than adding a new button, which we will do now. The button should be set to "run a script", and a new script function should be generated. Once that is done, this is the function we are going to write:

```
function Pick2_CommandExecuted(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      //declare variables
      var ppt1 : GwxDynamic =
      ThisConfiguration.GetDynamicObjectByName("PPT1");
      var ppt2 : GwxDynamic =
      ThisConfiguration.GetDynamicObjectByName("PPT2");

      //write value from ppt1 to ppt2
      ppt2.WriteValueToPrimaryDataSource(ppt1.GetValueOfPri
      maryDataSource());
}
```

As you may notice, the first part of the function is exactly the same as in the previous example. The last line of the body introduces the write function, called

`WriteValueToPrimaryDataSource`. This function takes as a parameter the value to write to its associated data source, which in this case is the other process point's value.

When you run this example, clicking on this new button will have the effect of copying the first process point's value to the second one.

3.5.3 Security Login Dialog Invocation through Scripting

In the GENESIS64 Core course, you learned two ways to invoke the Security Login dialog: from GraphWorX64 and from the Workbench. A third way of invoking the login dialog is through scripting.

To use the scripting approach, add a button to your GraphWorX64 display with the RunScript command. Then, double-click the RunScript event and do not check the box for Add a new form to display the Script Editor. Next, add the following code to the empty function body:

```
ThisWindow.ShowLoginDialog();
```

This function has only one line of code, which is the one needed to show the Security Login dialog. When going into Runtime Mode, and pressing the button, you should see the Security Login dialog come up as shown in Figure 3.9

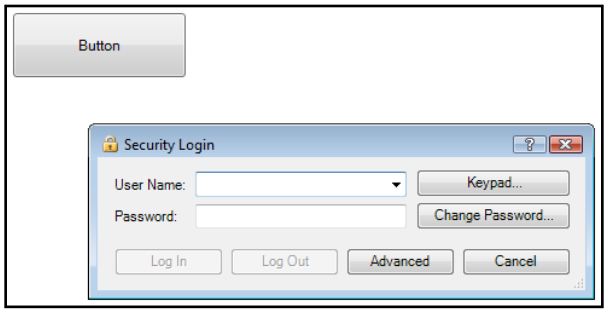


Figure 3.9 - Invoking the Security Login Dialog from Scripting

Lab: Basic Scripting 3

3.1 Lab Overview

Now that we have completed most of the Graphics for icTonics, they would like us to build some logic into their displays so that when the tank is full, the valves will automatically shut off to prevent any overflows.

Estimated time to complete this lab: 20 minutes

3.1.1 Task Checklist

In this lab, you will complete the following tasks:

- Understand the logic behind the "Login" button
- Add logic to simulate tank level
- Add logic to shutoff valves when the tank is full

3.1.2 Support Files

You will need the following files to complete this lab:

- icTonicsHome.gdfx
- icTonicsTankMonitor.gdfx

3.2 Understanding the Login Button

In this chapter, you learned how to bring up the login dialog. This is exactly what we used in the icTonicsHome.gdfx. In this section of the lab, we will look at the code behind the button.

1. Locate the icTonicsTankMonitor.gdfx and double-click to open it.

2. Open the Script Editor and pin it. If you do not see the Script Editor on your display, check the Show/Hide section of the View tab and make sure that Scripts is checked.

3. In the ThisDisplayCode module, you should see a function called “Login”. In this function, there is one line of code that brings up the login dialog that you seen when you click on it.

4. It is very easy to write some script to achieve what you need.

3.3 Simulate Tank Level

In this section, we are going to use a simple script to simulate our tank level data.

1. Open the icTonicsTankMonitor.gdfx.

2. Add a checkbox to the display; change the Command to Run Script.

3. Click the lightning bolt in the Dynamics Explorer and give the name “icTonics_SimulateData” as the name of the script. Press the enter key when you are done.

4. In the Script Options dialog that comes up, just click on "OK". This should bring up the Script Editor and you should be in a module called icTonicsCode with a function called icTonics_SimulateData.

5. In this script, we are going to read in the water valve and syrup valve status. If the valves are open, we are going to increase the tank level. We have shown you the code of how to read and write to a process point during the lecture portion; therefore, we will not show you the exact code here. But you can refer to the pseudo code below for the logic, if you are not sure, refer to the lecture or the API Reference Guide.

```
function icTonics_SimulateData(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      /* declare three variables for water valve, syrup
      valve, and Tank level */
      var water, syrup, tankLevel
      water = get object with water valve status
      syrup = get object with syrup valve status
      tankLevel = get object with tankLevel status

      If waterValve is open then tankLevel = tankLevel+1
      If syrupValve is open then tankLevel = tankLevel+1
}
```

Hint: You can drill into your popup menu to get the valve status value. You do not need to add any additiona process points on the display.

6. Once you tested that your code works, change the CheckBox execution trigger to ExecuteWhileChecked and set the WhileCheckedInterval to 1000. Also, set the Dynamic Behavior InitialClick to True. This will allow us to run the script every second updating the value at this interval.

7. Go into runtime and see that the checkbox is automatically checked. Open your valves and see that the tank fills up.

8. We will now hide this checkbox on the display so that the operators will not see it.

   Hint: Remember the eye in the Explorer?

9. As one last step, we should add some logic to the code so that when the tank reaches 100%, it will close the valve automatically.

```
function icTonics_SimulateData(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      /* declare three variables for water valve, syrup
      valve, and Tank level */
      var water, syrup, tankLevel
      water = get object with water valve status
      syrup = get object with syrup valve status
      tankLevel = get object with tankLevel status

      If tankLevel is not 100%{
            If waterValve is open then tankLevel =
            tankLevel+1
            If syrupValve is open then tankLevel =
            tankLevel+1
      }
}
```

10. Go into runtime and test your display.

NOTE: You do not need to write scripts to achieve what we did in this lab. You can use a checkbox with expressions to get the same results. We did it with a script here to give you some practice.

Scripting Basics Reviewed *R3*

3.1 Objectives Review

After completing this chapter, you should know how to:

- Use the Script Editor in GraphWorX64
- Use GraphWorx64 global variables
- Manipulate OPC tags through Scripting
- Insert objects and change properties through Scripting
- Launch the Security Login through Scripting

3.2 Summary

GENESIS64 includes a script editor that allows you to write your own code to achieve extra functions. The language used is Jscript.Net, which belongs to the .Net programming language family. Knowledge in this field will certainly help when you begin to create your scripts.

Although we allow you to achieve special tasks with scripts, you should consider it as a last resource. GENESIS64 includes many different features; you should always explore the natively available features before you consider using a script to achieve the task.

When you begin creating your scripts, there are a couple helpful tools. We have created an API Reference Help document that you can access from the Start menu. It contains all the API documentation with the list of classes, methods, properties and events, as well as their description, parameters, remarks, etc. Furthermore, the script editor contains a built-in IntelliSense functionality. As you script, IntelliSense will show you the available methods and the parameters required.

As you script, it is useful to understand the different global variables that can obtain GraphWorX32 objects and properties. These variables are ThisConfiguration, ThisDocument, and ThisWindow.

Throughout this chapter, we have shown you scripting examples in using a message box, reading and writing to an OPC point, and bring up dialogs such as the Security Login dialog. You can use these as the basis for more complicated scripts

3.3 Terms

IntelliSense	JScript.NET	ThisConfiguration
ThisDocument	ThisWindow	

3.4 Test Your Knowledge

In this chapter, we discussed a few basic concepts of scripting within GraphWorX64. Before you move on to the next chapter, please be sure that you can answer the following questions.

1. Which programming language does GraphWorX64 uses for scripting?

   a. VisualBasic.NET
   b. C#
   c. JScript.NET
   d. VB Script

2. Which of the following is <u>NOT</u> one of the global variables available in GraphWorX64

   a. ThisDisplay
   b. ThisConfiguration
   c. ThisDocument
   d. ThisWindow

3. True or False: The only tool available to help you with scripting is the API Reference Guide that comes with the installation.

4. Which function can you use to read a simulated point localism::var1 on a GraphWorX64 display? Let's assume that we have already declared a variable called Point that refers to object on the display.

   a. OPC.read("localism::var1")
   b. Point.Value()
   c. Point.GetValueFromDynamicObject()
   d. Point.GetValueOfPriamryDataSource()

5. Which function can you use to reference a button on a GraphWorX64 display?

   a. ThisConfiguration.GetObjectByName()
   b. ThisDocument.GetObjectByName()
   c. ThisConfigurartion.GetDynamicObjectByName()
   d. ThisDocument.GetDynamicObjectByName()

6. True or False: You can add more references to your script.

7. True or False: You cannot create any forms for your scripts. You can only use GraphWorX64 popup windows as your forms.

Scripting Advanced *4*

4.1 Objectives

After completing this chapter, you should know how to:

- Use the API reference to look up properties or methods available
- Insert an object in runtime mode
- Manipulate an object's property in runtime mode
- Create a custom form
- Interact with a database through Scripting

4.2 Introductions

In the previous chapter, we introduced you to some basic concepts of scripting with JScript.NET in GraphWorX64 and provided you with examples to perform some basic functions like reading the value of a process point with scripting.

In this chapter, we will continue our discussion on scripting and expand to include several different things that you can do with scripting. We will also go through a few ways that the API reference can help you in your endeavor to add scripts to your project.

4.3 Manipulating Objects in Runtime Mode

Typically, you configure your displays, save them, and simply run them. However, there are certain times that you may want to change an object's property or even insert objects onto the display in runtime mode. There is no native way to do this within GraphWorX64 while you are in runtime mode.

Although through scripting, it *is* possible to alter the appearance of certain objects and insert them in runtime mode. In this section, we will demonstrate how you can quickly alter or add an object to a display in runtime mode. One thing you

should keep in mind while you make configuration changes to your display in runtime mode is that displays cannot be saved in this mode. If you wish to save the changes that you make during runtime, you must change back to configuration mode and then save your display.

4.3.1 Changing Object Properties in Runtime Mode

Once you have a 2D object on your display, it is then possible to manipulate the object while in runtime through scripting. In the following example, you will see how to change the color of an ellipse and then how to hide it.

As usual, once you have inserted an ellipse onto the display in configuration mode, you will then need to give it a name, as shown in Figure 4.1. Then place a button on the display to run a script that will change the Fill color of the ellipse.

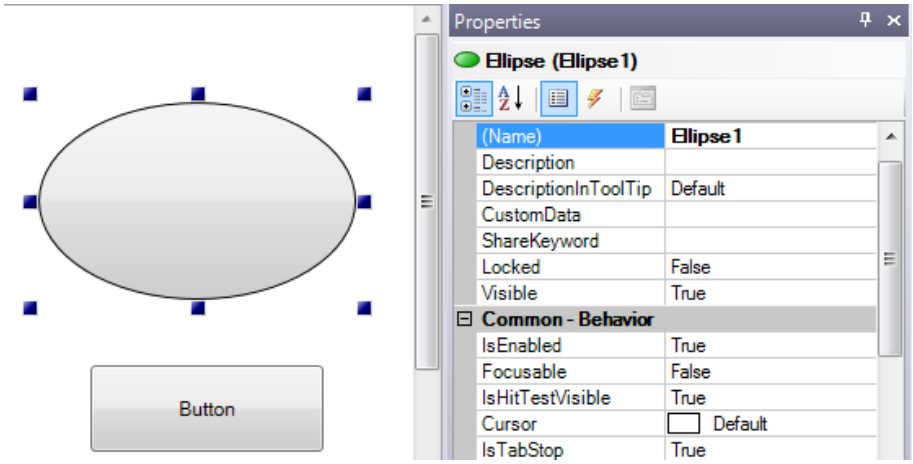


Figure 4.1 - Naming the Ellipse in Configuration Mode

Now you need to write a script that will change the Fill color of the ellipse as well as make it into a circle. The first line of code will create a GwxEllipse variable and will get a reference to the ellipse itself. The second line will change the Fill property of the ellipse, and the last line, sets the horizontal radius equal to the vertical radius, which will change it into a circle. The resulting code should resemble the following:

```
var ellipse : GwxEllipse =
ThisConfiguration.GetObjectByName("Ellipse1");
ellipse.Fill = Brushes.HotPink;
ellipse.RadiusX = ellipse.RadiusY
```

Lastly, you should enter runtime mode and test your script. You should see the ellipse's Fill color change to the one defined in the script when pressing the button. The result should look similar to Figure 4.2.

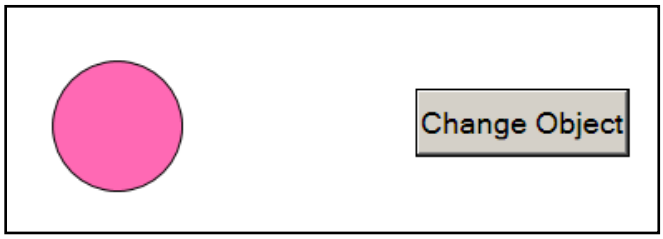


Figure 4.2 - Changed Ellipse Properties

Obviously, there are many more properties to GwxEllipse than the Fill, RadiusX, and RadiusY. You can find all these properties in the API Reference documentation. Just look under the GwxEllipse Members. This would contain all the Methods and Properties for this type of objects. This applies to other classes as well, such as the GwxRectangle, which we will look at next.

4.3.2 Inserting Objects on a Display

The way you usually insert an object into a display is by the commands found in the ribbon. However, you can also do this with scripting. In this example, you will be inserting a blue rectangle into your GraphWorX64 display while in Runtime mode.

We start by creating a new display, where we will place a button whose pick action is "Run Script". Instead of double-clicking on the event name to generate a default-named function, we will use "CreateRectangle" as the function name and then press enter to go to the Script Editor to edit this new function.

In the script, we need to declare the Rectangle object, create the object including its size and color, and then display the object. This can be achieved in the following way:

```
//declare variable
var myRectangle : GwxRectangle;

// set rectangle properties
myRectangle = GwxRectangle.Create(ThisConfiguration,
                                  new
Rect(50,50,200,100),
                                  ThisConfiguration.Root,
                                  false,
                                  null);
// change rectangle color
```

```
myRectangle.Fill = Brushes.Blue;
```

Once you have entered this code, save your display and enter runtime mode to test your button. You should see a blue rectangle on your display as shown in Figure 4.3.

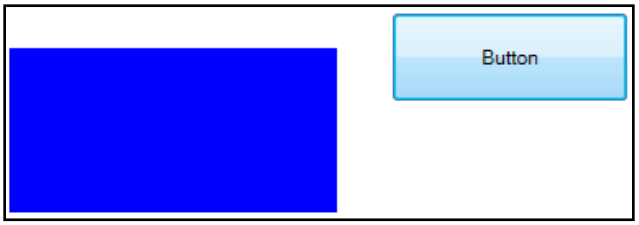


Figure 4.3 - Rectangle Created in Runtime Mode

You will see that when you switch back to Configuration Mode, the new rectangle is still there. This is the intended behavior and can be used to generate your display through scripting. Suppose, for example, that you have to draw a lot of objects placed in a well known way. You can automate this by writing a script that will draw all the objects for you, then go back into Configuration Mode and simply use the newly created objects.

4.3.2.1 The Code Digested

Let us take a closer look at the code written here. First, we declared a variable of the type GwxRectangle. You may wonder at this point, how you would know which type to use. The simplest way is to look inside your *GENESIS64 API Reference* document. In here, it lists all of the classes available inside the GraphWorX64 namespace. This means that if you are looking for a certain class like the GwxRectangle, it would be listed there along with a description. Although there are many different classes available, they are named intuitively. Generally, the name of the class is Gwx*ObjectType*, where *ObjectType* is the type of the object such as Rectangle, Ellipse, Dynamic, etc. If you are not sure of the object type, you can always add the object you wish to use first into your GraphWorX64 display, and look at the Object Explorer, it will tell you the type of the object.

The next line allows you to create a rectangle with specific properties. If we look at the reference guide, you will find the declaration for the function as shown in Figure 4.4. As you type in the function in your script editor, you should also see IntelliSense which guides you to the different parameters you need.

```
JScript
public static function Create(
        config : GwxConfiguration,
        dimensions : Rect,
        parentElement : IGwxElementContainer,
        selectAfterCreate : boolean,
        initializePropertiesFrom : GwxElement
) : GwxRectangle
```

Figure 4.4 - GwxRectangle.Create Declaration

There are five parameters here. First, it is looking for a GraphWorX64 Configuration. In this case, we can use the variable ThisConfiguration to satisfy this criterion. Secondly, it is looking for a dimension of the rectangle. In this case, we are using a structure inside the .NET framework library; the name of the structure is called **Rect**, which takes the left, top, width, and height of the rectangle as numbers (Doubles to be exact) respectively.

The rest of the parameters are intuitive from the naming convention. It is looking for a parent element of the rectangle, whether to select the rectangle after we create it, and a location to initialize the properties of the rectangle. In the case of our example, the parent is the root of the display since there is no other object on the display. We get to the root by using ThisConfiguration.Root. Then, since we do not want to select the object after creating it, we give a false for the fourth parameter. Finally, it asks for a GraphWorX64 element, but since we cannot derive from this class as specified in the API Reference, we will give it a null object. This completes the creation of the rectangle.

The last line allows us to set the color for the rectangle. As you type, you will probably notice a window that lists many different colors under the Brushes class. Feel free to choose your favorite color from the list as opposed to the blue that we used in our example.

4.3.3 GraphWorX64 Shapes

As you may have noticed in the previous examples that both GwxEllipse and GwxRectangle have the property Fill, which allows you to specify the color that fills the object. There are many more Methods and Properties that are common amongst the different GraphWorX64 shapes.

If you examine the API Reference document closely, you will notice that under the GwxEllipse Class and GwxRectangle Class declarations they both inherit another class call GwxShape. This is where these two classes obtain their shared

properties and methods. Since there are shared properties and methods, knowing how to use a method or property for one type of GwxShape means you know how to use it for all of the shapes.

This concept applies to all shapes within GraphWorX64. Now, you may ask, what is considered a GwxShape? Well, if you take a look at the Shapes section in the Home tab of the ribbon, you will find most of your shapes in there. Line, Arc, Spline, and Polyline may seem unrelated to shapes at a glance (after all, they are line segments). But, if you drill down far enough, you will notice that these eventually all inherit the GwxShape class.

This concept does not stop with shapes; it can apply to other objects too. For example, you can look at the GwxProcessPoint class. As you know, a process point is a dynamic, and if you drill down enough, you will find a class called GwxDynamic. And if you know how to use the DataSource properties on a process point object, chances are, you know how to use the same property on any other dynamics, like a color dynamic since they inherit the same base class.

4.4 Creating a Custom Dialog Using a Form

Sometimes it may be useful to create your own form for some specific task. For example you may want to prompt the user to enter some information (i.e. his name), or you want to ask for a confirmation for some specific action. In these cases, you can take advantage of the Script Editor to create and edit a form.

First, let us start with a brand new display, where we add a button. We set this button's pick action to "Run Script" and we double click the "RunScript" event to have it generates the function for us. However, this time, we will choose to create a new form.

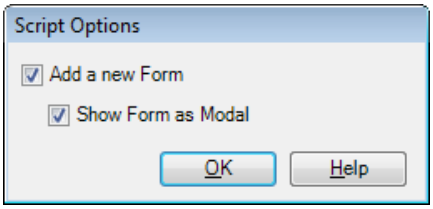


Figure 4.5 - Creating a new Form

We can also choose if we want the form to be modal. In this case, we want it to be modal, because we do not want the user to do any other action before they confirm the information present in the form.

After clicking the "OK" button, you will see that in the Script Editor a module and a form have been added. If you click the ThisDisplayForm on the left-hand pane in the editor, you will see something similar to Figure 4.6.

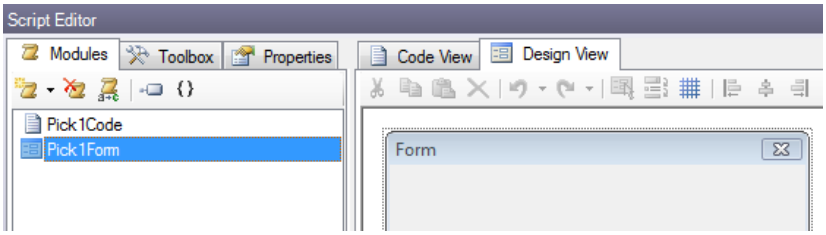


Figure 4.6 - The Form in the Script Editor

The form is empty now and therefore, we need to design it. You can click on the Toolbox tab to find the available controls. From there, you can add controls to the form in the same way it works in Microsoft Visual Studio. To edit the controls' properties you can click the Properties tab.

In this example, we will add a label, containing our message, and some radio buttons to allow the user to select one of the available options. The form should look like Figure 4.7.

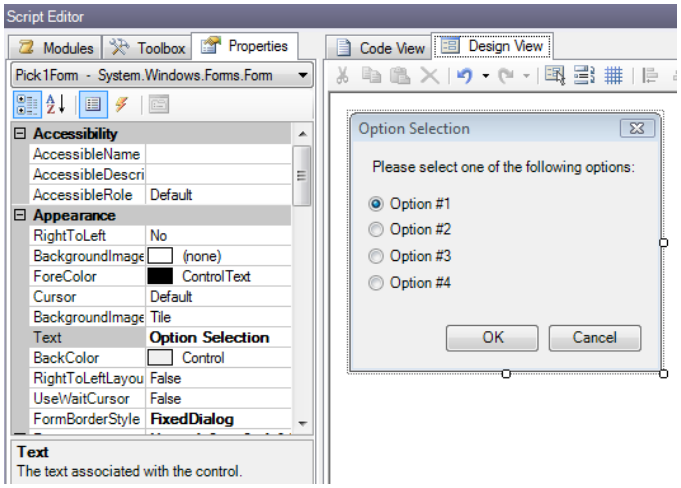


Figure 4.7 - The Final Form

The generated code in this case is a bit more complex than what we worked with in the previous examples, but is nothing too difficult if you have some basic code experience in the .NET environment.

What we need to do now is to create the handler for the two button's Click events, which will actually dismiss the dialog. Moreover, we want to create a variable where to host the selection value. Let us start from the Selection property. In the form code module, just after the Pick1Form declaration we add our property:

```
// Pick1Form
public class Pick1Form
{
      public var Selection : int;
    [...]
```

This property's value will be 1, 2, 3 or 4 depending upon which option has been chosen (note that we have the first option as the default one). The value will be 0 if we press the Cancel button.

You should define the two event handler functions in the following way:

```
public function buttonOK_Click(sender : System.Object, e :
System.EventArgs)
{
      if (radioButton1.Checked) this.Selection = 1;
      else if (radioButton2.Checked) this.Selection = 2;
      else if (radioButton3.Checked) this.Selection = 3;
      else this.Selection = 4;

      form.DialogResult =
      System.Windows.Forms.DialogResult.OK;
      form.Close();
}

public function buttonCancel_Click(sender : System.Object,
e : System.EventArgs)
{
      form.DialogResult =
      System.Windows.Forms.DialogResult.Cancel;
      form.Close();
}
```

The logic is simple; we just set our property value, define the dialog result, and close the form.

NOTE: Do not forget to associate the code with the button if you just insert the code directly.

The form itself would be useless if we do not call it and bring it up in a script module. The code to call and show the form has been auto-generated as part of the process and we only need to integrate it to actually use what we defined above. The full function to bring up the form by clicking the button is as follows:

```
function Pick1_CommandExecuted(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var form : Pick1Form = new Pick1Form();
      var result : System.Windows.Forms.DialogResult =
      ThisDocument.ShowModalScriptForm(form.Form);
      var selection : int = form.Selection;
      MessageBox.Show("Result is: " + result + ", Selection is: "
      + selection);
}
```

In the function body, the first two lines were automatically generated by the system for us, while the last two lines were added by us. What this code does is to instantiate and show the form, getting back its DialogResult value. After this, we define a variable where to read the form's Selection property and then we finally show the result and the selection to the user. In a real application, you would have used this information to take some action, rather than showing it to the user.

Now that we have completed the creation of our example, we can go into Runtime Mode and see what happens.

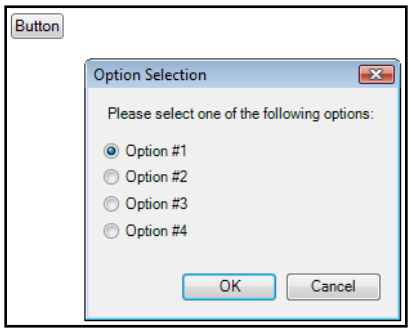


Figure 4.8 - The Form Shown at Runtime

Clicking the button shows the form above. Suppose that we pick the third option and then click OK, this is what we get, something similar to what's shown in Figure 4.9.

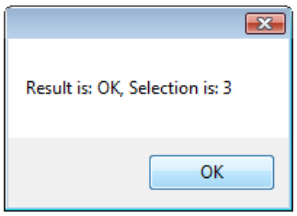


Figure 4.9 - The Result of Clicking OK on the Form

If we press Cancel instead, with any selection, we should get a dialog similar to the one shown in Figure 4.10.

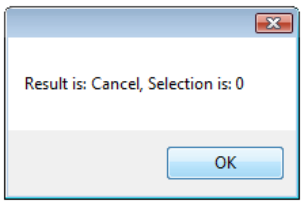


Figure 4.10 - The Result of Clicking on Cancel on the Form

As you can see, the form can have any possible appearance that you decide to create. This greatly extends the capabilities of the software, because you can write your own code to handle the logic behind the form.

> **NOTE**: It is also possible to create modal popup dialogs by using GraphWorX64 tools. For example, you can have a different display file, which is shown as a modal popup for the currently opened display. In this "native" popup, you can take advantage of all the GraphWorX64 features, including 3D, trends, aliases, etc. Therefore, before creating your new form from scratch you may want to consider first if using the tools provided with the product is enough for your needs.

4.5 Reading Data from a Database

Other than interacting with GraphWorX64 and the display, scripting technology allows you to work with the "external" .NET world. In this example, we will take advantage of the .NET technology to establish a connection with an existing database, and will display the date inside it.

First, you need to have a database manager installed, either SQL Server or whatever you prefer, provided you can connect to it using .NET.

To prepare for this example, we need to create a new database, with at least one table filled with some data. Of course, you may prefer to use an already existing database (i.e. NorthWind), and this is perfectly fine with this example, it will be just a matter of adjusting the connection string and the query used to retrieve your data out from the database.

In our case, we are going to create a new database, which we call "ScriptingTestDb", and a new table called "Positions", which will contain some sample data. Figure 4.11 displays the structure of the table we created.

Column Name	Data Type	Allow Nulls
▶ ObjectName	nvarchar(50)	☑
X	decimal(18, 0)	☑
Y	decimal(18, 0)	☑
Z	decimal(18, 0)	☑
		☐

Figure 4.11 - The Database Structure

Figure 4.12 shows the content we put into it.

	ObjectName	X	Y	Z
▶	obj1	0	0	0
	obj2	1	2	3
	obj3	2	2	2
*	NULL	NULL	NULL	NULL

Figure 4.12 - The Database Content

When the database and table creation is complete, we can start GraphWorX64 and prepare it for retrieving data.

4.5.1 Displaying Database Content in a Label

In a new GraphWorX64 display, add a button to the display, where we will write the code needed for this example. Prepare the button to execute a script, and create the function to be called when the button is pressed.

We should also create a Label, which we use to display the contents from the database. This is, of course, only necessary for this example. In your case, you should display or use the data coming from the database depending on your needs. The GraphWorX64 display may look similar to Figure 4.13

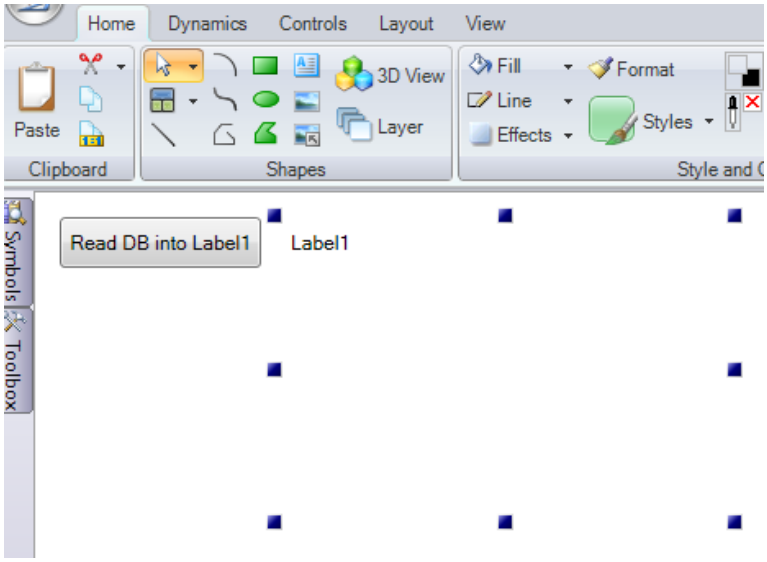


Figure 4.13 - The GraphWorX64 Display

Now, we are going to edit the script function that will do the work as follows:

```
var label : Label =
ThisConfiguration.GetObjectByName("Label1").ToDependencyObj
ect();
var connString : String = "Data Source=(local);Initial
Catalog=ScriptingTestDb;Integrated Security=SSPI";
var conn : SqlConnection = new SqlConnection(connString);
var reader : SqlDataReader;

try
{
      conn.Open();
      var command : SqlCommand = new SqlCommand("SELECT *
      FROM Positions", conn);
      reader = command.ExecuteReader();
      var content : String = "";
      while (reader.Read())
      {
            content += reader["ObjectName"] + " (" +
            reader["X"] + ", " + reader["Y"] + ", " +
            reader["Z"] + ")\n";
      }
      Reader.Close();
      label.Content.Text = content;
}
catch(ex : Exception) {
      MessageBox.Show (exMessage);
}
```

You can now go into runtime mode and test your display. After pressing the button, you should get something similar to Figure 4.14.

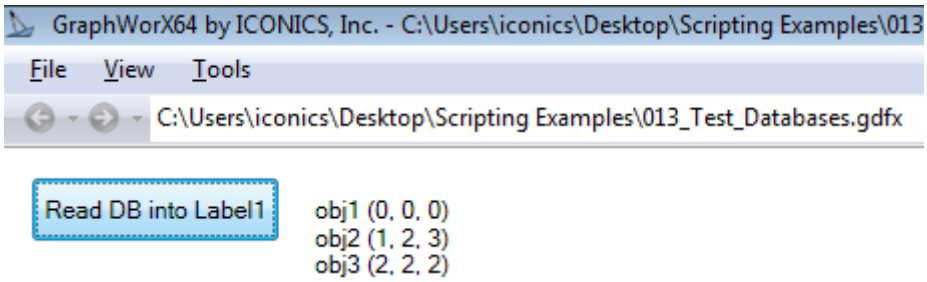


Figure 4.14 - The Effect of Running the Script

The script writes the content of the database into the label. This completes our example. We will analyze the code and take a closer look next.

4.5.1.1 The Code Digested

In the code example, we first declared a few variables that we need. First, we get a reference to the label where we will output the information we obtain from the database. Then, in the next line, we create the database connection string. We will not go into too much detail in how to construct a database connection string. You can use the one in the code as an example and change the necessary information. If you are truly not sure, create a UDL file, connect to the database, and look at the All tab. That should contain the information you need to construct the string; use the previous code as a guideline.

Finally, we create a connection object that uses our connection string and a reader that we will use to scan the results of a query.

> **NOTE:** In the example code, we have added System.Data.SqlClient in the namespaces list. If you did not do the same, you will run into issues and errors will be shown at the bottom of the Script Editor. You can either add the namespace or use the full declaration.

The next section is a try and catch block. We use this in an attempt to prevent possible problems, and if there are such problems, we *catch* them instead of letting it crash.

The try block contains the "meat" of the code. It first opens the connection to the database. After opening the connection, we create a command for SQL to execute. In here, you would simply put an SQL query that returns the data set you wish to see. Our command simply retrieves all the records in the Positions table. At this point, you can consider the reader variable as a two dimensional array that looks just like a table.

After executing the command, we also prepare a variable to store the database content temporarily, before writing it into the label.

We now use a while block to iterate over each record. If it makes it easier to visualize, you can consider reader a one dimensional array, containing only the current record. We then use the column name as the index of this array to retrieve the value in a particular column. You can alternatively use the zero-base index number if you wish.

Through each iteration, we output the content in a custom format: *obj (x, y, z)* and then we put a line return at the end so that the next record would start on a new line. When we have read all the records returned by the SQL query, we close the reader and transfer the content to the label for display.

4.5.2 Display Database Content in the Data Grid Form

At times, you may simply want to display the data in a table as oppose to writing it to a label or a process point. You can do this with the help of a windows form. The form we are looking for is called *DataGridView*, which comes with .NET.

> **NOTE:** If you do not see the DataGridView, you can add it by looking for the System.Windows.Forms.dll usually located in C:\Windows\Microsoft.NET \Framework64\v2.0.50727. This will add a few more .NET forms into your Toolbox.

With this form on our display and a button that will load the content of a table for us when pressed, we are all set to start the scripting. The code is similar to the previous example in many cases. We have to make a connection to the database, run an SQL command to get the content of the table, and then fill the grid with the data return.

Except this time, we are going to breakdown the code into two functions. We will actually use a helper function to load our data into the grid. Let us examine the following code:

```
function Database_ReadValues(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
	var commandString : String;
	commandString = "SELECT * FROM Positions";
	PopulateGrid(commandString);
}

function PopulateGrid (commandString)
{
	//declare and initialize variables for DB connection
	var connString : String = "Data Source=(local);Initial
	Catalog=ScriptingTestDb; Integrated Security=SSPI";
	var connection : SqlConnection = new
	SqlConnection(connString);
	var command : SqlCommand = new SqlCommand(commandString,
	connection);
	var dataAdapter : SqlDataAdapter = new
	SqlDataAdapter(command);;
	var dataTable : DataTable = new DataTable();

	// get data and fill the table
	dataAdapter.Fill(dataTable);

	//declare and initialize variables for controls
	var wfCtl : GwxWindowsFormsControl =
	GwxWindowsFormsControl(ThisConfiguration.GetObjectByName("D
	ataGridView1"));
	var grid : System.Windows.Forms.DataGridView =
	System.Windows.Forms.DataGridView(wfCtl.Control);

	// fill the grid with data
	grid.DataSource = dataTable;
}
```

NOTE: Remember that we have added System.Data.SqlClient as a namespace. If you did not perform that step, your code should use the full namespace for the SQL functions we use.

Once you have the above code in your display, you can go into runtime and give it a try. The results should look similar to Figure 4.15.

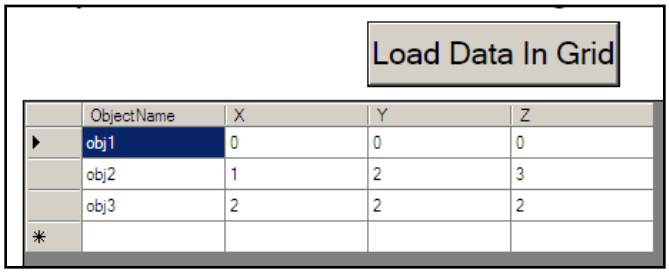


Figure 4.15 - Positions Table in DataGridViewer

4.5.2.1 The Code Digested

In the main function, Database_ReadValues, there is very little that we did. We created a command string, set the value of that string to the desire SQL query, and we call the PopulateGrid function, giving our SQL query.

Most of the work is done in your helper function: PopulateGrid. In here, similar to the previous example, we created a connection string and used it to make a connection to our database. We then created a SqlCommand, using the command string we passed in and the database connection. To execute this SqlCommand, we use a SqlDataAdapter and give it the SqlCommand we constructed. The final step of the first section, we create a DataTable. We will use this to store the data information.

Once all the variables are in place, we call the dataAdapter to execute the command and fill the dataTable with the data returned by the command.

Now that we have the data, we need to put the data into our control. The third section of the code declares the variables necessary to get the control from the display and then creates a handle to the control. The last step, we set the data source of the control to our dataTable, which contains the content of the SQL table.

Lab: Viewing Trend Data in a Table *4*

4.1 Lab Overview

While we can display our trend data in a chart, icTonics upper management would like to see the data in spreadsheet format. In this lab, we will use a bit of scripting in GraphWorX64 in conjunction with SQL commands to give icTonics a table of trended data.

Estimated time to complete this lab: 20 minutes

4.1.1 Task Checklist

In this lab, you will complete the following tasks:

- Find and insert the Microsoft DataGridView form
- Script with the DataGridView form
- Show trend data in the DataGridView form

4.1.2 Support Files

You will need the following to complete this lab:

- icTonicsTemplate.gdfx
- Logged Trend Data

4.2 Trend Data in a Grid View

In this chapter, we covered some basics on how to read data from a database and display it. In this lab, we will try to apply a similar technique to show data logged in our TrendWorX64 logging database.

1. Create a new GraphWorX64 display with the icTonicsTemplate.gdfx.

2. Add the DataGridView and a button to load the data through script. You can call the script "icTonics_RefreshTrendData".

3. In the script, we are going to connect to the TrendWorX64 logging database and pull out data from it. For now, you can hard code the time range in which you want to view data. Again, we went through the necessary steps on how to do this in the lecture and will not provide you with the exact code. However, you can use the pseudo code below as a guideline.

```
function icTonics_RefreshTrendData(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      // declare necessary variables and set it
      var connString
      var connection
      var commandString
      var command
      var dataAdapter
      var dataTable

      // get data and fill the table
      dataAdapter.Fill(dataTable);

      //declare and initialize variables for controls
      var wfCtl
      var grid

      // fill the grid with data inside the table
      grid.DataSource = dataTable;
}
```

NOTE: If you feel that using a helper function would help you later on, feel free to create it now. Also, you can choose any available trend table for now.

4. When you are done, after clicking the Refresh Data button, your table should look similar to Figure L4.1.

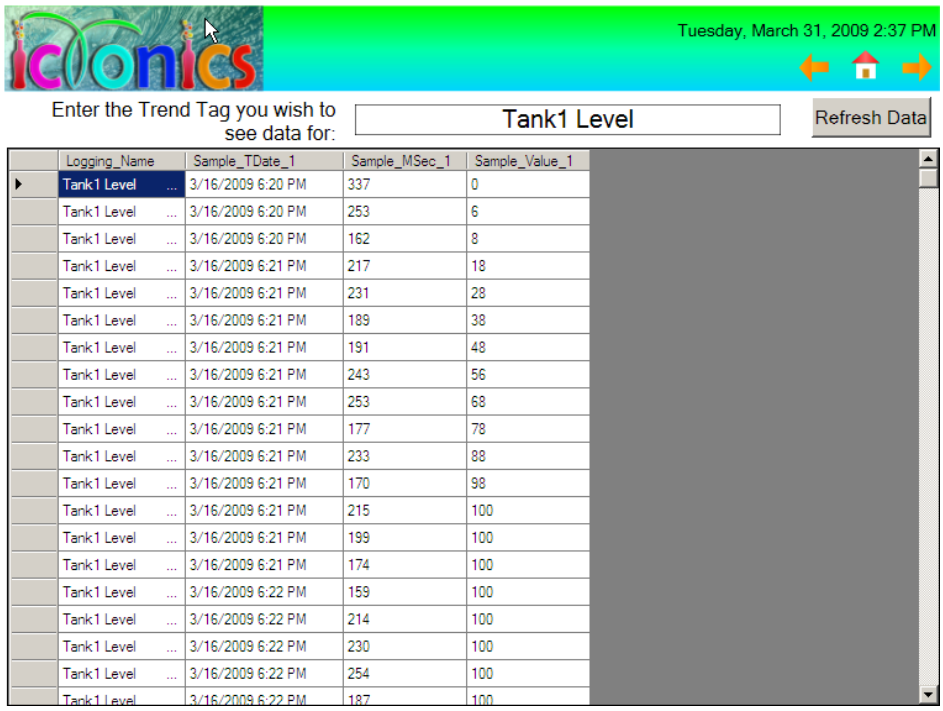
Signal_Index	Earliest_TDate	Latest_TDate	Rec_Modified	Fill_Index	Sample_TDate_1	Sample_MSec_1	Sample
1	3/16/2009 6:20 ...	3/16/2009 6:23 ...	0	36	3/16/2009 6:20 ...	337	0
1	3/16/2009 6:23 ...	3/16/2009 6:26 ...	0	36	3/16/2009 6:23 ...	178	6
1	3/16/2009 6:26 ...	3/16/2009 6:29 ...	0	36	3/16/2009 6:26 ...	156	6
1	3/16/2009 6:29 ...	3/16/2009 6:32 ...	0	36	3/16/2009 6:29 ...	182	6
1	3/16/2009 6:32 ...	3/16/2009 6:35 ...	0	36	3/16/2009 6:32 ...	189	6
1	3/16/2009 6:35 ...	3/16/2009 6:38 ...	0	36	3/16/2009 6:35 ...	246	10
1	3/16/2009 6:38 ...	3/16/2009 6:41 ...	0	36	3/16/2009 6:38 ...	175	53
1	3/16/2009 6:41 ...	3/16/2009 6:44 ...	0	36	3/16/2009 6:41 ...	251	90
1	3/16/2009 6:44 ...	3/16/2009 6:47 ...	0	36	3/16/2009 6:44 ...	169	14
1	3/16/2009 6:47 ...	3/16/2009 6:50 ...	0	36	3/16/2009 6:47 ...	177	-1.5923
1	3/16/2009 6:50 ...	3/16/2009 6:53 ...	0	36	3/16/2009 6:50 ...	132	1
1	3/16/2009 6:53 ...	3/16/2009 6:56 ...	0	36	3/16/2009 6:53 ...	210	33
1	3/16/2009 6:56 ...	3/16/2009 6:59 ...	0	36	3/16/2009 6:56 ...	138	34
1	3/16/2009 6:59 ...	3/16/2009 7:02 ...	0	36	3/16/2009 6:59 ...	184	34
1	3/16/2009 7:02 ...	3/16/2009 7:05 ...	0	36	3/16/2009 7:02 ...	127	34
1	3/16/2009 7:05 ...	3/16/2009 7:08 ...	0	36	3/16/2009 7:05 ...	128	34
1	3/16/2009 7:08 ...	3/16/2009 7:11 ...	0	36	3/16/2009 7:08 ...	175	34
1	3/16/2009 7:11 ...	3/16/2009 7:14 ...	0	36	3/16/2009 7:11 ...	112	34
1	3/16/2009 7:14 ...	3/16/2009 7:17 ...	0	36	3/16/2009 7:14 ...	159	34
1	3/16/2009 7:17 ...	3/16/2009 7:20 ...	0	36	3/16/2009 7:17 ...	190	34

Figure L4.1 - Data in TankLevel_1 in a DataGridView Form

5. For these purposes, once you can view the data in the database, it is good enough. However, if you want to give yourself a challenge, you can write a more complex SQL query so that instead of looking at the signal index, it returns the actual tag name and instead of listing the values in its original format, lists it in chronological order down the table per tag.

6. As one last task, you should allow the user to select which tag they want to see data for so that they are not bombarded with all the information at once. The user should be able to choose the one tag they are interested in. If you did the previous step, you can use the name of the tag; otherwise, you can use the tag index to pick the tag.

7. Your finished display in runtime should look like this:



Logging_Name	Sample_TDate_1	Sample_MSec_1	Sample_Value_1
Tank1 Level ...	3/16/2009 6:20 PM	337	0
Tank1 Level ...	3/16/2009 6:20 PM	253	6
Tank1 Level ...	3/16/2009 6:20 PM	162	8
Tank1 Level ...	3/16/2009 6:21 PM	217	18
Tank1 Level ...	3/16/2009 6:21 PM	231	28
Tank1 Level ...	3/16/2009 6:21 PM	189	38
Tank1 Level ...	3/16/2009 6:21 PM	191	48
Tank1 Level ...	3/16/2009 6:21 PM	243	56
Tank1 Level ...	3/16/2009 6:21 PM	253	68
Tank1 Level ...	3/16/2009 6:21 PM	177	78
Tank1 Level ...	3/16/2009 6:21 PM	233	88
Tank1 Level ...	3/16/2009 6:21 PM	170	98
Tank1 Level ...	3/16/2009 6:21 PM	215	100
Tank1 Level ...	3/16/2009 6:21 PM	199	100
Tank1 Level ...	3/16/2009 6:21 PM	174	100
Tank1 Level ...	3/16/2009 6:22 PM	159	100
Tank1 Level ...	3/16/2009 6:22 PM	214	100
Tank1 Level ...	3/16/2009 6:22 PM	230	100
Tank1 Level ...	3/16/2009 6:22 PM	254	100
Tank1 Level	3/16/2009 6:22 PM	187	100

Figure L4.2 - Finished icTonicsTrendReport Display

Scripting Advanced Reviewed *R4*

4.1 Objectives Review

After completing this chapter, you should know how to:

- Use the API reference to look up properties or methods available
- Insert an object in runtime mode
- Manipulate an object's property in runtime mode
- Create a custom form
- Interact with a database through Scripting

4.2 Summary

GraphWorX64 coupled with JScript.Net allows you superior control of your graphics even in runtime mode. You can change an object's property such as color, size, and others as well as add new objects to the display. It is important to remember that although you can manipulate your display during runtime, you can only save the display in configuration mode; anything you change in the display will be lost if you failed to save it in configuration mode before browsing away to another display.

While GraphWorX64 allows you to bring up child displays that can simulate a form, you can also create one using JScript. Forms can help you obtain operator inputs in a very specific format. For example, if you want to have a Dvorak on screen keyboard, you may want to create your own form since one may not be readily available.

When you associate forms with scripts, you will see a separate item in the Modules list for your form. It is usually has the word Form appended to the end of your script. You can use the Design View of the form to arrange the necessary details and then use the Code View to add logic behind objects on the form.

As the last exercise, we showed you how to retrieve data from a database and display it in a label and in a Windows Form: DataGridView.

4.3 Terms

Address Space	AlarmWorX64	Common Object Model
Configuration Mode	Distributed Control System	EarthWorX64
GenBroker	GenTray	GraphWorX64
GUI	HMI	HyperHistorian
IAC	OLE	OPC AE
OPC Client	OPC DA	OPC HDA
OPC Server	OPC	PLC
Project File	Runtime Mode	SCADA
Tags	Service Oriented Architecture	TrendWorX64
Unified Data Browser	Unified Data Manager	Workbench

4.4 Test Your Knowledge

In this section, we expanded on scripting topics. Before you continue to the next chapter, make sure you can answer the following questions.

1. Which of the following is something that you CANNOT change through scripting during runtime?

   a. A rectangle's color
   b. A line's thickness
   c. The data source of a process point
   d. All of the above
   e. None of the above

2. True or False: You can run a script in configuration mode to help you develop your display.

3. Which of the following namespace is NOT included by default?

   a. System.Data
   b. Ico.Gwx
   c. System.Windows.Input
   d. System.Windows.Forms

4. True or False: Because GraphWorX64 provides properties and methods that uses inheritancy, knowing how to use a property or method for one object type

means you should know how to use the same property or method for a different object type that inherits the same class.

5. True or False: You can output information from a database to a process point.

Scripting with Controls 5

5.1 Objectives

After completing this chapter, you should know how to:

- Obtain a reference to a GENESIS64 Control to work with it
- Interact with the following controls with Scripting
  - AlamrWorX64
  - TrendWorX64
  - EarthWorX64

5.2 Introductions

In GENESIS64, you may use a few different controls within GraphWorX64 and the Workbench. These include AlarmWorX64, TrendWorX64 and EarthWorX64. These controls expose an API to interact with them by using custom code, either JScript.NET scripts or custom code written in some external Dynamic Link Library (DLL).

In this chapter, we will focus on writing some simple scripts to demonstrate how to interact with these controls. For example, you will be able to:

- Acknowledge all the active alarms in AlarmWorX64
- Add a new pen to a series in TrendWorX64
- Create a pushpin and add it to EarthWorX64 map

These represent only a few possible examples. In order to proceed with this chapter and its examples you should be familiar with writing some simple scripts in JScript.NET from inside GraphWorX64. If you do not feel comfortable with that, please refer to the GraphWorX64 Scripting chapter to learn more about this technology.

5.3 Getting a Reference to GENESIS64 Controls

In order to interact with a GENESIS64 control, you need to perform a common operation: get a reference to the control. Since you will repeat this operation each time you need to work with a Control, it is a good idea to be familiar with this operation before continuing.

We start with an empty GraphWorX64 display where we add all of our Controls: an AlarmWorX64 Viewer control, an EarthWorX64 map, and a TrendWorX64 Viewer control. We also give a name to each of these objects, calling them "AwxControl1", "EwxControl1", "TwxControl1". You can of course use any other name you like for your controls. Our GraphWorX64 display looks like Figure 5.1.

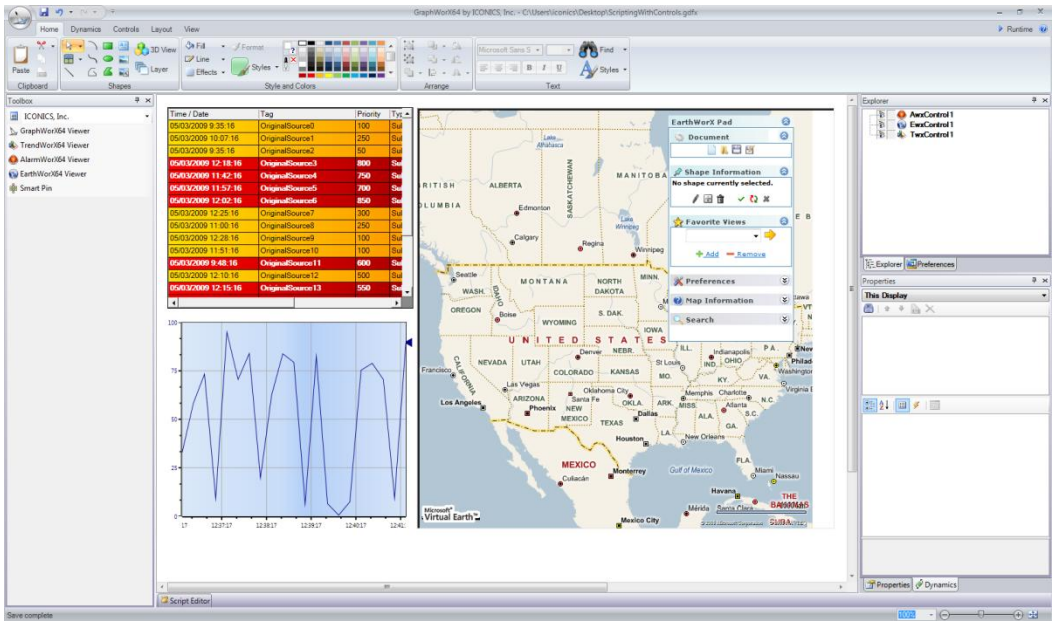


Figure 5.1 - GraphWorX64 Display with Three Controls

We will now add a button, which will trigger our script when pressed. When you add a new button, it will have by default the "Load Display" Pick Action associated to it; we will change its type to be "Run Script" instead.

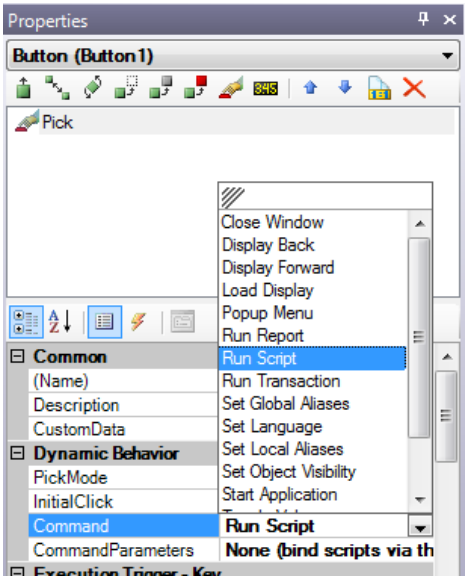


Figure 5.2 - A Button to Execute a Script

Now, from the Events tab we can double-click the "RunScript" event to let GraphWorX64 generate a script function for us. The display will now look like Figure 5.3:

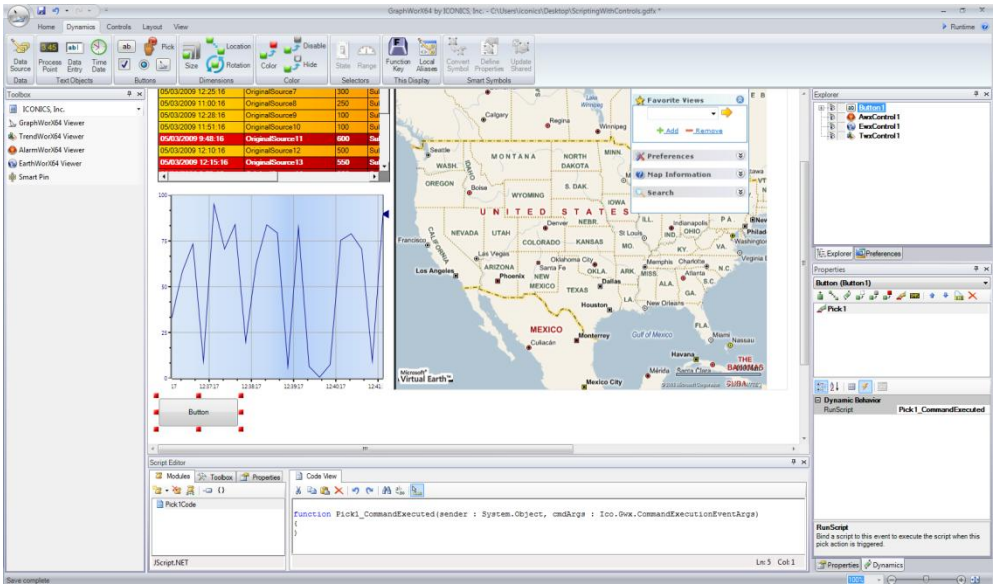


Figure 5.3 - The Display after Creating the Script Function

We can edit this function to get the reference to each of the single controls. This operation will be slightly different for EarthWorX64 compared to AlarmWorX64 and TrendWorX64. This is due to the fact that EarthWorX64 is a Windows Forms control, while AlarmWorX64 Viewer and TrendWorX64 Viewer are pure WPF Controls.

In order to get a reference to a WPF Control, like AlarmWorX64 Viewer, you need to insert the code below into your script function:

```
var awxViewer : Ico.Awx.AwxViewControl;
awxViewer =
ThisConfiguration.GetObjectByName("AwxControl1").ToDependen
cyObject();
```

The first line of this script code declares a variable called `awxViewer`, whose type is `Ico.Awx.AwxViewControl`. The second line actually gets the reference from the current display configuration. Note that we are passing the AlarmWorX64 Viewer Control's name. In case you chose a different name, or if you have multiple AlarmWorX64 Viewer instances on your display, you may want to adjust this parameter to match your settings.

> **NOTE:** Please also note that you can easily collapse this code to a single line. The way you write your code also depends on your coding style and preferences.

To get a reference to the TrendWorX64 Viewer is similar. The only difference is the type of the variable that you declare and the name of the control you assign to it. Here is the code for your comparison purpose with the differences highlighted:

```
var twxViewer : Ico.Twx.TwxViewControl;
twxViewer =
ThisConfiguration.GetObjectByName("TwxControl1").ToDependen
cyObject();
```

As you can see, there are only minor differences between the two script portions.

As we mentioned earlier, EarthWorX is just slightly different. This is due to the fact that EarthWorX is actually a Windows Forms controls, and for this reason, it is hosted into a "wrapper" control inside GraphWorX64. This means we first need to retrieve its wrapper, and then extract the control out of it. The script code to perform such operation is as follows:

```
var wfControl : Ico.Gwx.GwxWindowsFormsControl;
var ewxViewer : Ico.Ewx.EwxControl;
wfControl =
Ico.Gwx.GwxWindowsFormsControl(ThisConfiguration.GetObjectB
yName("EwxControl1"));
ewxViewer = Ico.Ewx.EwxControl(wfControl.Control);
```

As we did for the other two types of controls, we first declare the variables we need, which in this case are the wrapper and the EarthWorX Control. Then we get a reference to the wrapper first that contains the control, and using the wrapper, we get the control we are interested. This is similar to how we got a handle to the DataGridView control in the previous chapter.

Using each of these code snippets you will end up having one variable (`awxViewer`, `twxViewer` or `ewxViewer`), which can be used to access the various methods and properties exposed by these controls. For example, suppose the AlarmWorX64 Viewer control exposes a method called DoSomething, which takes no parameter, then you will be able to invoke this method using the following script instruction:

```
awxViewer.DoSomething();
```

To conclude this section, we will present the whole script function, which we created so far, even if it is still doing nothing else than getting a reference to the 3 controls we added on our display:

```
function Pick1_CommandExecuted(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      var awxViewer : Ico.Awx.AwxViewControl;
      awxViewer =
      ThisConfiguration.GetObjectByName("AwxControl1").ToDe
      pendencyObject();

      var twxViewer : Ico.Twx.TwxViewControl;
      twxViewer =
      ThisConfiguration.GetObjectByName("TwxControl1").ToDe
      pendencyObject();

      var wfControl : Ico.Gwx.GwxWindowsFormsControl;
      var ewxViewer : Ico.Ewx.EwxControl;
      wfControl =
      Ico.Gwx.GwxWindowsFormsControl(ThisConfiguration.GetO
      bjectByName("EwxControl1"));
      ewxViewer = Ico.Ewx.EwxControl(wfControl.Control);
}
```

You may also want to consider creating some function to avoid repeating this code many times in your application. For example, you can define a function like the one below:

```
function GetAwxViewer(controlName : string) :
Ico.Awx.AwxViewControl
{
      var awxViewer : Ico.Awx.AwxViewControl;
      awxViewer =
      ThisConfiguration.GetObjectByName(controlName).ToDepe
      ndencyObject();
      return awxViewer;
}
```

Then you can invoke the DoSomething method we already used earlier, in this way:

```
GetAwxViewer("AwxControl1").DoSomething();
```

5.4 Working with AlarmWorX64 Viewer

The first control we will work with is the AlarmWorX64 Viewer. We will use its API to perform the following tasks:

- Create a new tab with an alarm grid
- Toggle the Grouping feature
- Acknowledge all the alarms

5.4.1 Understanding the AlarmWorX64 Viewer Control

Before we start writing scripts for this control, we need to understand its structure. The AlarmWorX64 Viewer Control is a user control, which may contain one or more tabs. Each tab may contain one or more grids or charts. These elements (grid and chart) are called views. Often, you will need to work with these views to perform your tasks.

Suppose we only have one tab containing one alarm grid (the default scenario) in our AlarmWorX64 Viewer Control. To obtain a reference to a view, you can use the following code. This reference will allow you to access methods and properties related to the view.

```
var awxViewer : Ico.Awx.AwxViewControl;
awxViewer =
ThisConfiguration.GetObjectByName("AwxControl1").ToDependen
cyObject();

var view : Ico.Awx.DataGrid.AwxGridView;
view =
Ico.Awx.DataGrid.AwxGridView(awxViewer.Tabs[0].Views[0]);
```

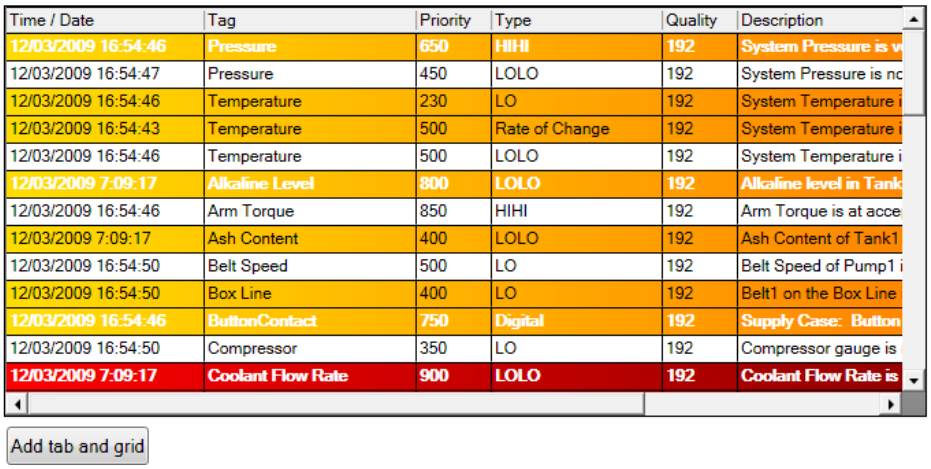
We already discussed the first block of this script to get a reference to the user control in the previous section. The last two lines are used to get a reference to the grid itself (whose type is `AwxGridView`), accessing the `Tabs` and `Views` arrays by using two indexes. Since we know we only have one tab with one grid in the default case, we used zero (0) for both indexes, as we only have one element. Needless to say, the indexing for the arrays is zero-based.

Now that we have the view object, we can interact with it, calling the grid's methods, or altering the properties.

5.4.2 Creating a New Tab within an Alarm Grid

As we said earlier, by default, the AlarmWorX64 Viewer only contains a single tab, with has one grid inside it. We can use the API to create a new tab at runtime dynamically. We can also populate that tab with one or more views.

In this example, we are going to add a new tab with a single grid in it. Suppose that our AlarmWorX64 Viewer looks like the one displayed in Figure 5.4 while in runtime mode.



Time / Date	Tag	Priority	Type	Quality	Description
12/03/2009 16:54:46	Pressure	650	HIHI	192	System Pressure is v
12/03/2009 16:54:47	Pressure	450	LOLO	192	System Pressure is nc
12/03/2009 16:54:46	Temperature	230	LO	192	System Temperature i
12/03/2009 16:54:43	Temperature	500	Rate of Change	192	System Temperature i
12/03/2009 16:54:46	Temperature	500	LOLO	192	System Temperature i
12/03/2009 7:09:17	Alkaline Level	800	LOLO	192	Alkaline level in Tank
12/03/2009 16:54:46	Arm Torque	850	HIHI	192	Arm Torque is at acce
12/03/2009 7:09:17	Ash Content	400	LOLO	192	Ash Content of Tank1
12/03/2009 16:54:50	Belt Speed	500	LO	192	Belt Speed of Pump1 i
12/03/2009 16:54:50	Box Line	400	LO	192	Belt1 on the Box Line
12/03/2009 16:54:46	ButtonContact	750	Digital	192	Supply Case: Button
12/03/2009 16:54:50	Compressor	350	LO	192	Compressor gauge is
12/03/2009 7:09:17	Coolant Flow Rate	900	LOLO	192	Coolant Flow Rate is

Figure 5.4 - The AlarmWorX64 Viewer Control in Runtime Mode

We want to have a button associated to a script, which will add a new tab containing an alarm grid to this control. To do that, we need to create an `Ico.WPF.Specialized.Tab` and an `Ico.Awx.DataGrid.AwxGridView` object, which represent the tab and the grid respectively. Once we have these two objects we will add the grid to the tab, and then the tab to the AlarmWorX64 Viewer control.

The source code that implements this behavior is as follows:

```
function NewTab(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var awxViewer : Ico.Awx.AwxViewControl;
      awxViewer =
      ThisConfiguration.GetObjectByName("AwxControl1").ToDe
      pendencyObject();

      // This script creates a new Tab with a grid inside
      var tab : Ico.WPF.Specialized.Tab;
      tab = new Ico.WPF.Specialized.Tab();
      tab.Name = "NewTab";
      tab.Views.Add(new Ico.Awx.DataGrid.AwxGridView());
      awxViewer.Tabs.Add(tab);
}
```

NOTE: In order for this code to work, you must add a reference for the Ico.WPF.Specialized library. You can find the IcoWPFSpecialized.dll in \Program Files\ICONICS\GENESIS64\Components directory.

As you can see, we have also set a name for the tab. Once you are finished with inserting the script, you can test it by going into runtime mode. Since we have associated the script to a button, once you go into runtime mode, click on that button, and your AlarmWorX64 Viewer will contain a new tab, as shown in Figure 5.5.

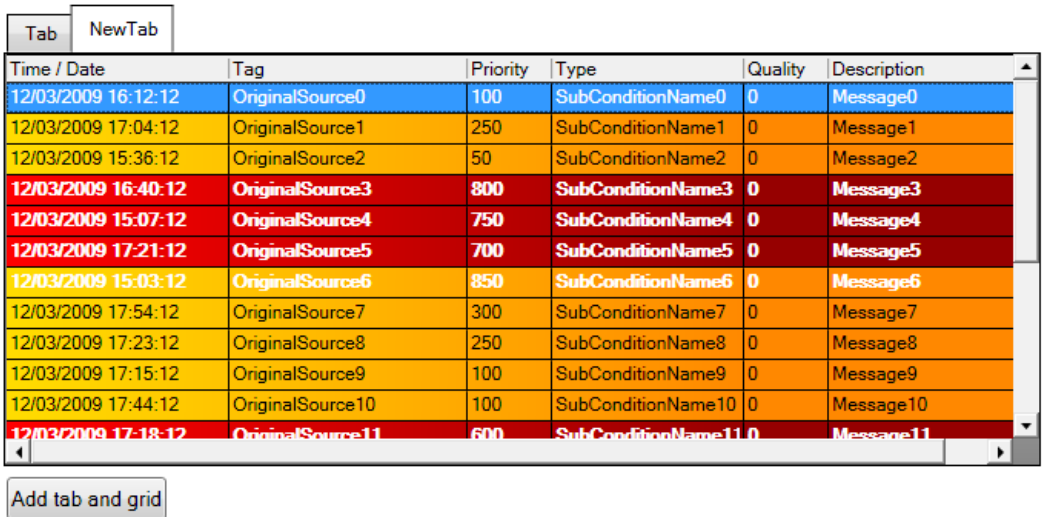


Figure 5.5 - Adding a New Tab to the Control

As you can see in Figure 5.5, there are actually two tabs in the control. The first one, called “Tab” is the one shown in Figure 5.4. When you only have one tab in the AlarmWorX64 Viewer, you do not see the tab label over the grid. These labels only appear when you have two or more tabs.

The configuration changes you perform during Runtime mode will persist when you go back into Configuration mode. In this way, you can automate the configuration of your display, by creating tabs and views for your AlarmWorX64 Control, for example, reading some configuration from a file or a database. Once you are all set with your changes, you can simply go back into Configuration mode and save your changes.

5.4.3 Toggling Grouping Feature

One of the many features available in the AlarmWorX64 Viewer is the capability to group your data in a grid depending on some of the columns. You can, for example, group your alarm based on severity or by area. This helps to organize your alarms and improves your understanding of the critical situations that may arise in your scenario.

You can control the grouping feature with scripting through the AlarmWorX64 Viewer API. The way this works is that each grid has a list of group objects. When this list is empty, no grouping is applied. If one or more groups exist, then the alarms display in clusters by taking these group one at a time, depending on their order. The group object's type is `Ico.Awx.DataGrid.AwxGroup` and these objects are contained in an `Ico.Awx.DataGrid.AwxGroupCollection` object.

In our example, we want to toggle the grouping feature. To do this we need to perform these two tasks alternately:

- Add a group to the collection
- Remove all groups from the collection.

The following function is used to achieve this:

```
function ToggleGroup(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var awxViewer : Ico.Awx.AwxViewControl;
      awxViewer =
      ThisConfiguration.GetObjectByName("AwxControl1").ToDe
      pendencyObject();

      // We get the grid
      var view : Ico.Awx.DataGrid.AwxGridView;
      view =
      Ico.Awx.DataGrid.AwxGridView(awxViewer.Tabs[0].Views[
      0]);

      // We just create a group using the column key
      if(view.AwxGroups.Count == 0)
```

```
            view.AwxGroups.Add(new
            Ico.Awx.DataGrid.AwxGroup("SeverityBaseEventType"));
            else view.AwxGroups.Clear();

            view.ReloadGroups();
        }
```

The first two blocks of code should be nothing new to you: we get a reference to the view (that is the grid). Then, we check to see if we are using groups in the viewer. If no groups are being used, the group count returns a zero (0); otherwise it would return the number of groups configured, which means we are already using this feature. Depending on the active use of this feature, either we add a new group or we clear them all to toggle the feature. The last line is used to reload the groups' configuration.

As you might notice, we use a string to specify which one is the column we intend to group by.

The following figure shows our alarms grouped by priority, after clicking the new button we have added and associated to the above script function.

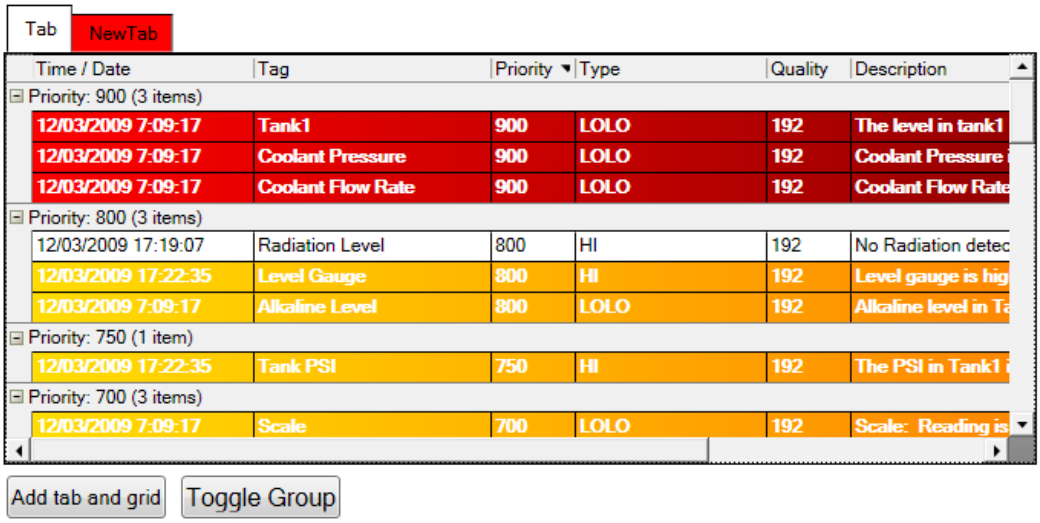


Figure 5.6 - Alarms Grouped by Priority

We have also sorted by priority descending so that we first see the groups of the most critical alarms. Clicking again on the Toggle Group button we have added to our display will ungroup the alarms.

5.4.4 Acknowledging All Alarms

The last example about scripting with AlarmWorX64 Viewer is about acknowledging alarms.

This task is quite simple to perform through scripting. You need to get a reference to the grid, as we did previously and then to call one of its API methods, called "Acknowledge". This method takes a parameter which is the type of acknowledge you intend to perform.

There are different types of acknowledgement available: you can acknowledge all the alarms, the selected ones, the ones currently filtered, or do a comparison to select a subset of the existing ones. In our case, we will simply acknowledge all the alarms. How you choose one of these acknowledge types is by mean of an enumeration called `Ico.Awx.DataGrid.AckType`.

The following is the function you need to define to acknowledge all of your alarms. As you can see, there is just one new line compared to the previous examples, and is the call to the Acknowledge method. Remember to associate this function with a button on your display.

```
function Acknowledge(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var awxViewer : Ico.Awx.AwxViewControl;
      awxViewer =
      ThisConfiguration.GetObjectByName("AwxControl1").ToDe
      pendencyObject();

      // We get the grid
      var view : Ico.Awx.DataGrid.AwxGridView;
      view =
      Ico.Awx.DataGrid.AwxGridView(awxViewer.Tabs[0].Views[
      0]);

      // We perform an ackknowledgement on all rows
      view.Acknowledge(Ico.Awx.DataGrid.AckType.All);
}
```

Once you go into Runtime mode, you should see several alarms coming. Clicking your new button will invoke the Acknowledge method that, after checking you have enough permission to perform this operation, it will acknowledge all the alarms in the specified grid.

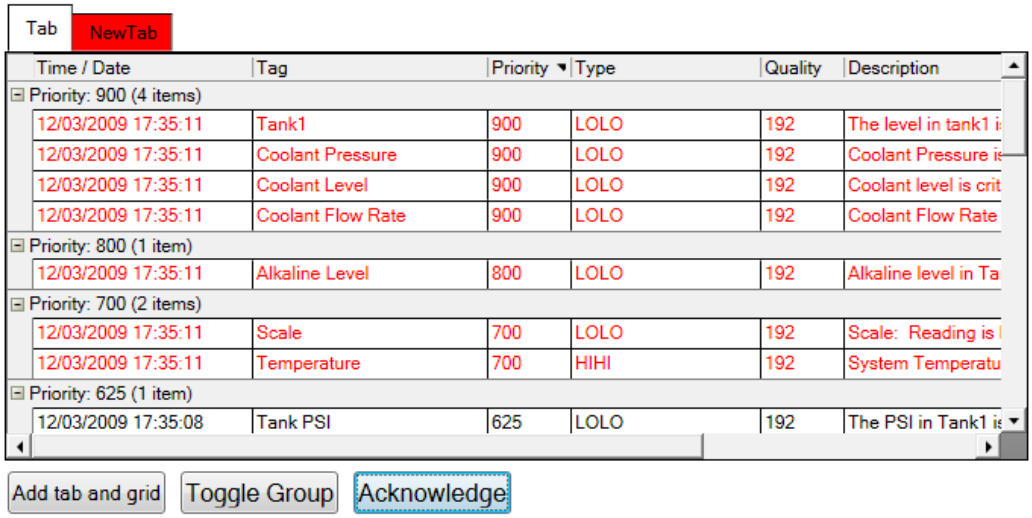


Figure 5.7 - All the Alarms have been acknowledged

As you can see from the red background in the other tab, we have only acknowledged alarms in the first tab. You can easily adapt this example to acknowledge all the alarms in all the grids and tabs.

5.5 Working with TrendWorX64 Viewer

We will now interact with the TrendWorX64 Viewer. Using its API, we want to do the following:

- Add a new pen
- Remove an existing pen
- Toggle the legend's visibility
- Swap the trend axis

5.5.1 Understanding the TrendWorX64 Viewer Control

The TrendWorX64 Viewer Control is conceptually very similar to the AlarmWorX64 Viewer Control we have discussed previously. If you skipped those sections, you may want to review the introductory section for the AlarmWorX64 Viewer to learn more about this type of user controls.

The TrendWorX64 Viewer Control uses the same structure based on tabs we have already seen in the alarm viewer component. Each tab may have one or more views. A view may contain one or more series, and finally, a series can draw trends with one or more pens. We need to know how to access these elements, because they are the ones we will use to work with the control.

Suppose we only have one tab containing one trend in our TrendWorX64 Viewer Control. The code below demonstrates how you can get a reference to the series so that you can then access its properties and methods:

```
var twxViewer : Ico.Twx.TwxViewControl;
twxViewer =
ThisConfiguration.GetObjectByName("TwxControl1").ToDependen
cyObject();

// Get the default Series
var series : Ico.Twx.TwxSeries;
series =
Ico.Twx.TwxChartComponent(twxViewer.Tabs[0].Views[0]).Serie
s[0];
```

The first lines of this script are something we are already familiar with: they just get a reference to the control. The last two lines get a reference to the first available series in the trend. Please note the use of indexes to access the various elements. You can alter these if you need to access a different tab, view, or series.

Now that we have the series object, we can interact with it, calling its methods, or altering its properties.

5.5.2 Adding a New Pen

The first example of script we create for the TrendWorX64 Viewer is the one that will let us add a new pen to an existing trend.

Although this example is a little bit longer than the ones we have seen thus far, it is not too complex. We first need to get a reference to the series we need to work with, and this is something we already introduced. Then, we need to create a pen and a data configuration for it. We will add this new pen to the series, and finally, we will force a restart of the trend to let the changes become active.

The function below will perform the above tasks:

```
function AddPen(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var twxViewer : Ico.Twx.TwxViewControl;
      twxViewer =
      ThisConfiguration.GetObjectByName("TwxControl1").ToDe
      pendencyObject();

      // Get the default Series
      var series : Ico.Twx.TwxSeries;
      series =
      Ico.Twx.TwxChartComponent(twxViewer.Tabs[0].Views[0])
      .Series[0];

      // Create a new pen
      var pen : Ico.Twx.TwxChartPen;
      pen = new Ico.Twx.TwxChartPen();

      // We use the pen counter to give the pen a name
      pen.Title="Pen #" + (series.Pens.Count+1);

      // Create the data configuration for the pen
      var configuration = new
      Ico.Data.DataConnector.DataConnectorConfiguration();
      configuration.ConnectorId = "SimulatorConnector";
      configuration.PointName =
      "localsim:sine::::::"+((series.Pens.Count+1)*10)+"000
      :100";
      pen.DataConnectionConfiguration = configuration;

      // Add the new pen to the series
      series.Pens.Add(pen);

      // Force an update of the runtime mode for the view
      Ico.Twx.TwxChartComponent(twxViewer.Tabs[0].Views[0])
      .IsRuntimeModeEnabled = false;
      Ico.Twx.TwxChartComponent(twxViewer.Tabs[0].Views[0])
      .IsRuntimeModeEnabled = true;
}
```

NOTE: In order for the above code to work, you should add a reference to the IcoDataConnector.dll located in \Program Files\ICONICS\GENESIS64 \Components directory.

The pen is an `Ico.Twx.TwxChartPen` object. We set its title using the series' pen counter, so that the pens will always have a valid name. If you specify the same name for more than a pen, you will still be able to add them to the trend, but you will get a "pen error" from the TrendWorX64 Viewer.

The definition of the data source is made in a way such that the first pen will use a sine signal with a 10 seconds period, the second pen will have 20 seconds period, then 30 seconds and so on.

> **NOTE:** The syntax for a local simulation point should be: localism:*mode:(name):DataType:LowLimit:HighLimit:InitialElapseTime:Duration:UpdateRate*
> In the case of our example, we only specified the mode, duration, and update rate.

The last two lines are used to stop and restart the runtime mode for the specified trends. If you do not do this, the changes will not apply until you go back to Configuration mode, then again into Runtime mode.

In our example, we have created a display with a TrendWorX64 Viewer with a single pen, and then we have added a button set to run the above script. This is what we have at the beginning:

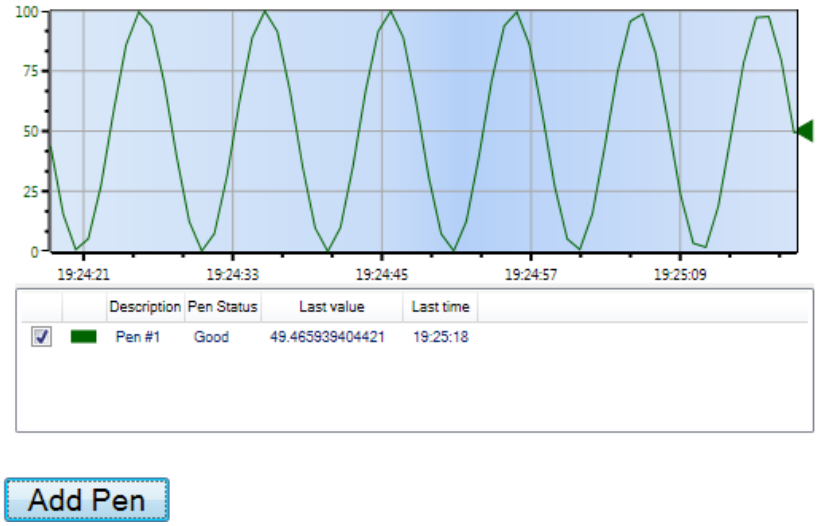


Figure 5.8 - A Single Pen in the Trend

After pressing the button once, a new pen is added and you see something similar to Figure 5.9.

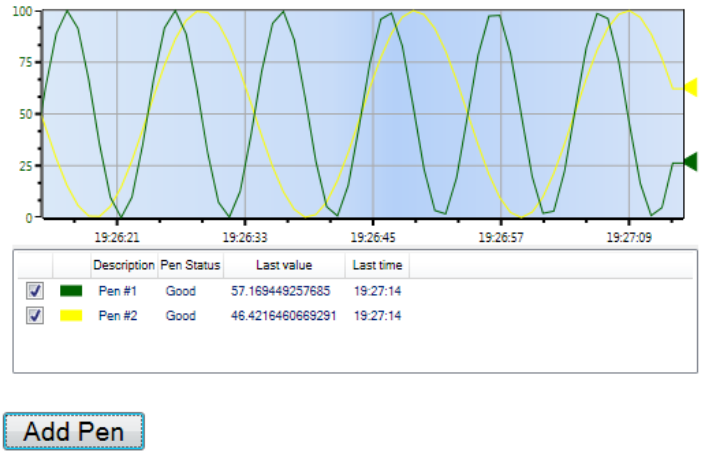


Figure 5.9 - Another Pen Added through Scripting

You are not restricted on the number of pens you can add to the control. In Figure 5.10, we added a few other pens, for a total of ten pens on the same trend at the same time.

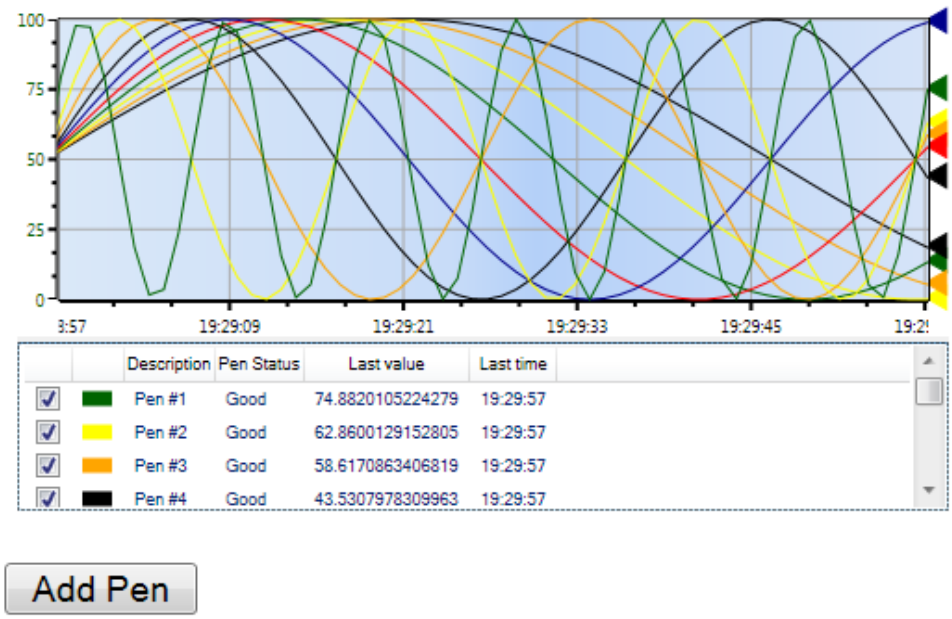


Figure 5.10 - A Trend with Ten Pens

This can be a quick way to automate the configuration of your control.

An interesting exercise would be to create a form that allows the user to enter some parameters, such as the pen's name, color, and thickness, for the new pen. Then, using this information, create the pen.

5.5.3 Removing an Existing Pen

Now that we have added many pens to our control, we also need a way to remove them. The way you remove a pen from a series is by calling the "RemoveAt" method of the Pens collection of the series. You will need to specify the pen position in the collection as a parameter. In our case, to keep the example simple, we will remove the last added pen. This we can easily achieve this using the pen counter for the series.

The function to remove the last added pen is as follows:

```
function RemovePen(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var twxViewer : Ico.Twx.TwxViewControl;
```

```
        twxViewer =
        ThisConfiguration.GetObjectByName("TwxControl1").ToDe
        pendencyObject();

        // Get the default Series
        var series : Ico.Twx.TwxSeries;
        series =
        Ico.Twx.TwxChartComponent(twxViewer.Tabs[0].Views[0])
        .Series[0];

        // Remove the last pen in this series
        if (series.Pens.Count != 0)
              series.Pens.RemoveAt(series.Pens.Count - 1);
}
```

The source code is rather simple, especially if you compare it with the code we wrote to add a new pen. The only important thing is to make sure we do not try to delete a pen if there is none in the collection. We check this by verifying the pen counter is not zero.

Also, note that in this case, we do not need to stop and restart the Runtime mode for the trend: the pens are removed immediately.

Starting from the previous example, we will press the new button to remove a pen for five times, to end up with a trend drawing with only five pens in it.

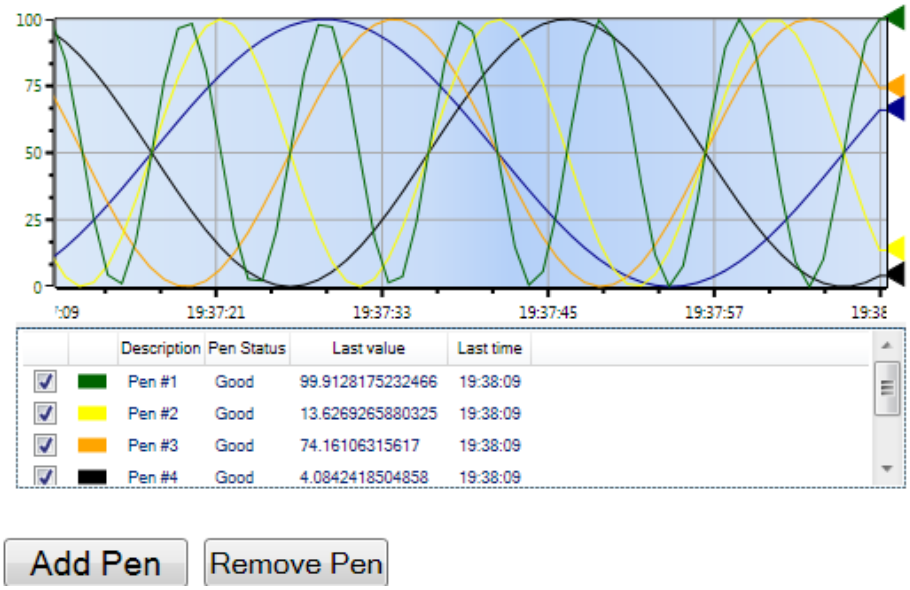


Figure 5.11 - A Trend where We Removed Five Pens

A good exercise would be to create a better user interface with a combo box containing the names of the existing pens to let your user select a specific pen to delete.

5.5.4 Toggling Legend Visibility

The TrendWorX64 Viewer can show a legend with some information about the currently drawn pens. You have a few examples of it in the screenshots above. That legend can be hidden/shown using the API as well. This can be done accessing a property of the trend viewer.

The property you need to modify is called `ShowLegend` and takes a Boolean value. Since we want to toggle its value, we will simply negate its current value. The function we have defined to do this is below:

```
function ChangeLegendVisibility(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      var twxViewer : Ico.Twx.TwxViewControl =
      ThisConfiguration.GetObjectByName("TwxControl1").ToDe
      pendencyObject();

      //Get the default View & change the legend visibility
      var view =
      Ico.Twx.TwxChartComponent(twxViewer.Tabs[0].Views[0])
      ;
      view.ShowLegend = !view.ShowLegend;
}
```

We just need to add a new button to our display, and hook it to this script function. When we go into runtime mode, and click on it, we will see the legend disappear, as in the following figure.

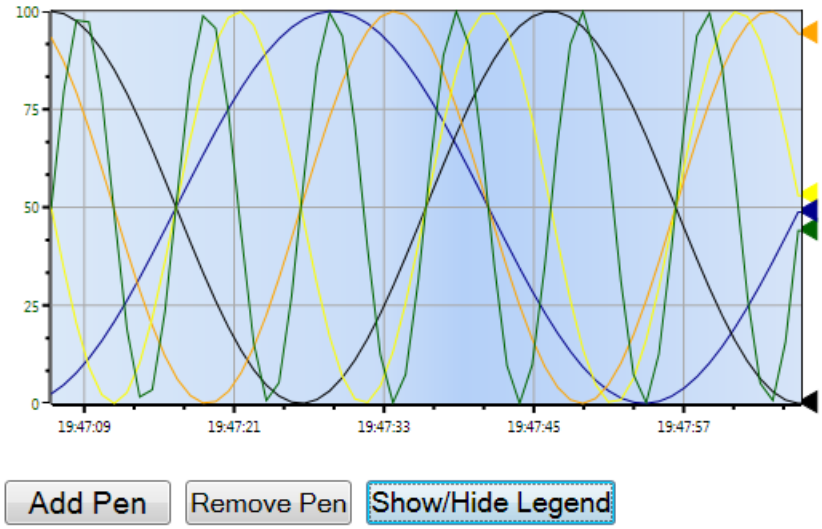


Figure 5.12 - Hiding the Legend

As for removing the pen, we don't need to restart the trend. If you click once more on the new button we created, the legend will go back to its place.

5.5.5 Swapping the Trend Axis

The last TrendWorX64 Viewer API example we present here is about swapping the trend axis. Normally, the pens draw the new values on the right side, so that the trend moves from right to left. You can swap this and have the trend moves from top to bottom, with pens drawing on the top of the trend.

To do this, we need to work with another simple Boolean property called `SwapAxes`. Setting this property to true will change the orientation for our trend.

The function we need to write is almost identical to the one we used to toggle the trend legend. We just need to change the property whose value we will be toggling.

```
function SwapAxis(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
        var twxViewer : Ico.Twx.TwxViewControl;
        twxViewer =
        ThisConfiguration.GetObjectByName("TwxControl1").ToDependen
        cyObject();

        // Get the default View and change its axis rotation
        var view =
        Ico.Twx.TwxChartComponent(twxViewer.Tabs[0].Views[0]);
        view.SwapAxes = !view.SwapAxes;
}
```

We add another button to the display and set its function to the one we just wrote. Going into runtime and pressing the button, will have the following effect on the trend.

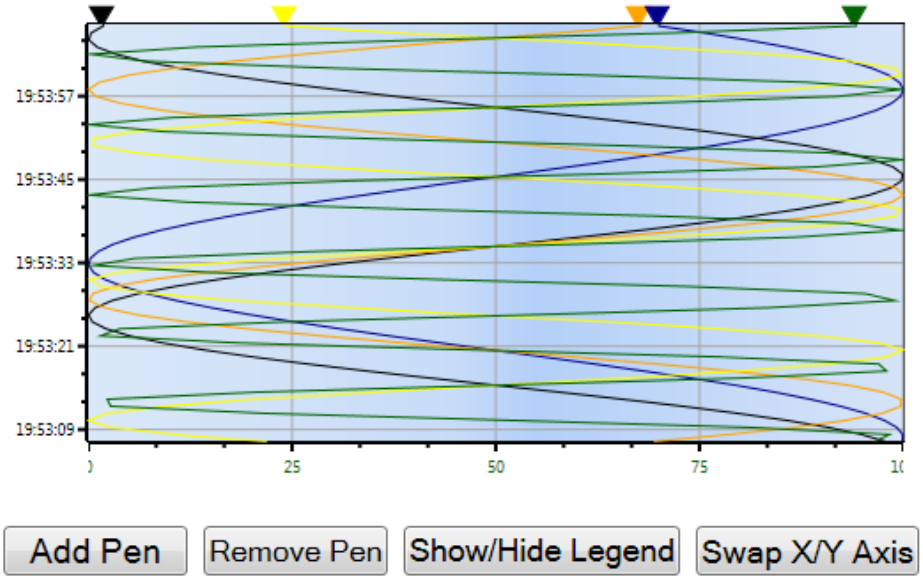


Figure 5.13 - Swapped Axis

The trend has been rotated by 90 degrees. Please note that we are still hiding the legend here. Now, we could click the "Show/Hide Legend" button to show it below the trend. Swapping axis will not alter the legend position.

5.6 Working with EarthWorX64 Control

The last control we will work with in this chapter is EarthWorX. We are going to use its API to perform these tasks:

- Create a pushpin and place it on the map
- Navigate between different locations on the map
- Improve the navigation system

5.6.1 Understanding the EarthWorX64 Control

The EarthWorX64 Control is a little bit different from the two other controls we worked with in this chapter. Technically speaking, it is not a WPF control; it is a Windows Forms control. For this reason, the way we interact with it is slightly different. Moreover, this control is not organized with tabs as AlarmWorX64 and TrendWorX64 viewers, but it has a single view containing the map object. The map itself can contain other objects such as Pushpins, Smartpins, and others.

First, we need to prepare our display to work with the control. We can continue working with the same display we have been using for the chapter, or we can create a new one. The important thing to remember is to name the EarthWorX control in the display. You can choose any name you like for it. We have chosen "EwxControl1".

To interact with EarthWorX API, we also need to add a reference to the Script Editor. This is because EarthWorX API is contained in a separate DLL, which is normally not referenced by your displays unless you need that.

To add a reference to the API DLL you need to click the "Edit References" button.

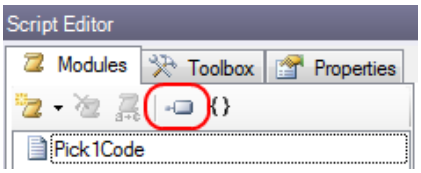


Figure 5.14 - The Edit References Button

From the window that appears, we click the button on the right end of the first free row, and we browse for the EwxApi.dll file, usually located in the \Program Files\ICONICS\GENESIS64\Components directory. Once we have done that, your references dialog should look similar to the one in Figure 5.15.

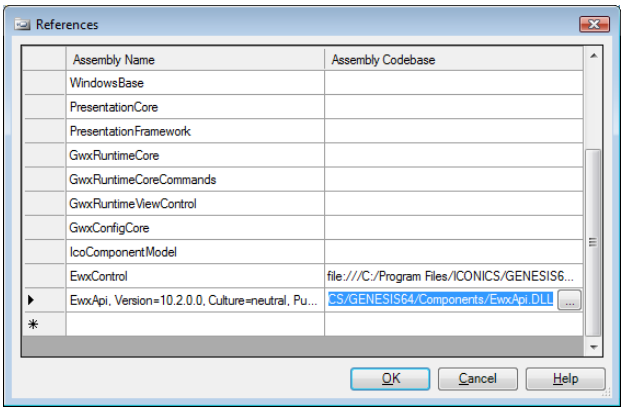


Figure 5.15 - The API Reference Dialog

In the previous sections, we have shown how to get a reference to the controls we want to work with. It is now time to recall how to get a reference to the EarthWorX control. The code for doing that is as follows:

```
var wfControl : Ico.Gwx.GwxWindowsFormsControl;
var ewxViewer : Ico.Ewx.EwxControl;

wfControl =
Ico.Gwx.GwxWindowsFormsControl(ThisConfiguration.GetObjectB
yName("EwxControl1"));
ewxViewer = Ico.Ewx.EwxControl(wfControl.Control);
```

In the above code snippet, we first get the `Ico.Gwx.GwxWindowsFormsControl` object, which contains the EarthWorX control. Note that we are using the name we chose for our control, if you have used a different name you need to adjust the code to fit your example.

Then we extract the `Ico.Ewx.EwxControl` object from the wrapper control. Note that the EarthWorX control is retrieved from the `Control` property of the `GwxWindowsFormsControl`.

You probably remember that for AlarmWorX64 Viewer we got a reference to an alarm grid to interact with it, while for the TrendWorX64 Viewer we got a reference to a series. For EarthWorX Control, we need to get a reference to the map instead. This can be done using the following code:

```
var map : Ico.Ewx.Objects.Map;
map = ewxViewer.GetMap();
```

The first line defines an object called `map` whose type is `Ico.Ewx.Objects.Map`. The second line gets a reference to the map from the EarthWorX Control, by means of the `GetMap` function.

Now that we have the map object, we have all that we need to start working with EarthWorX Control API. However, since getting a reference to the map is quite a common task, we want to define a utility function which we will be using throughout all the examples to speed up our work.

You have probably seen in the previous sections that we were often duplicating code. Although it is desirable to keep things simple in a training manual, we now want to show you a good practice to optimize your code and to reduce the possibility to introduce bugs in it.

Consider the following function:

```
function GetMap(controlName : String)
{
      var wfControl : Ico.Gwx.GwxWindowsFormsControl;
      var ewxViewer : Ico.Ewx.EwxControl;
      wfControl =
      Ico.Gwx.GwxWindowsFormsControl(ThisConfiguration.GetO
      bjectByName("EwxControl1"));
      ewxViewer = Ico.Ewx.EwxControl(wfControl.Control);

      var map : Ico.Ewx.Objects.Map;
      map = ewxViewer.GetMap();

      return map;
}
```

As you can see, it contains nothing more than the code we introduced in this section, but this code is wrapped in a function, called GetMap, which you can reuse in your code from now on. This function takes the name of the EarthWorX Control as a parameter, to make it more generic. If you need to use this in multiple displays, you can simply copy it over and just call it and pass the correct name of the control.

In our case, we will work with it using this code:

```
var map : Ico.Ewx.Objects.Map = GetMap("EwxControl1");
```

This is all that we need to do to obtain a map object in our script. You will see that this approach will be used in all the following examples.

5.6.2 Creating a Pushpin

EarthWorX is a very useful tool to visualize your assets on a map. For this reason, the first scripting example we propose shows how to put a simple object on the map.

We have already seen how to get a reference to the map object. To add an object to the map, you just need to call the map's method "AddShape", passing it the object to put on the map. The method is called "AddShape" because a shape is the generic name for any object you can add to the map.

To create a new object, in this case a Pushpin, we need to declare it, and then to invoke one of the available constructors to get the object. The code to do that is simple and we can quickly write a function that will add a new Pushpin to the map. We have decided to put it just over the Empire State Building in New York City, a well known place.

```
function AddPushpin(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var map : Ico.Ewx.Objects.Map =
      GetMap("EwxControl1");

      // Creates a pushpin
      var pushpin : Ico.Ewx.Objects.Pushpin = new
      Ico.Ewx.Objects.Pushpin("The Empire State Building",
      "The Empire State Building is a 102-story Art Deco
      skyscraper in New York City at the intersection of
      Fifth Avenue and West 34th Street. Its name is
      derived from the nickname for the state of New
      York.", 40.7483, -73.9857);
      map.AddShape(pushpin);
}
```

Our function consists of three simple commands.

The first one is used to get a reference to the map, as we have seen in the introductory section.

The second defines and initialize an `Ico.Ewx.Objects.Pushpin` object. We have used a special version of its constructor, which takes four parameters, but you can choose another one. IntelliSense will be very helpful here, as you can see from the following figure.

```
new Ico.Ewx.Objects.Pushpin("The Empire State Building", "The Emp
▲7 of 8▼ Pushpin(String title,String description,Double latitude,Double longitude)
```

Figure 5.16 - IntelliSense for Pushpin Creation

Following IntelliSense directions, we have specified a title, a short description and the geographical position of our landmark.

The third command used in this function simply calls the "AddShape" function passing our newly created Pushpin.

We can now associate this function to a button, save our display, and go into runtime mode. Figure 5.17 shows how our display looks before pressing the button, which will invoke the script to add the Pushpin.

Figure 5.17 - The Initial Map

After clicking the button, we will see a new object on the map. Moving the mouse over it will show us a tooltip, containing the information we have specified.

Figure 5.18 - A Pushpin Added to the Map

We can then interact with the map by zooming into the city and even change the view mode to verify if our object is correctly placed.

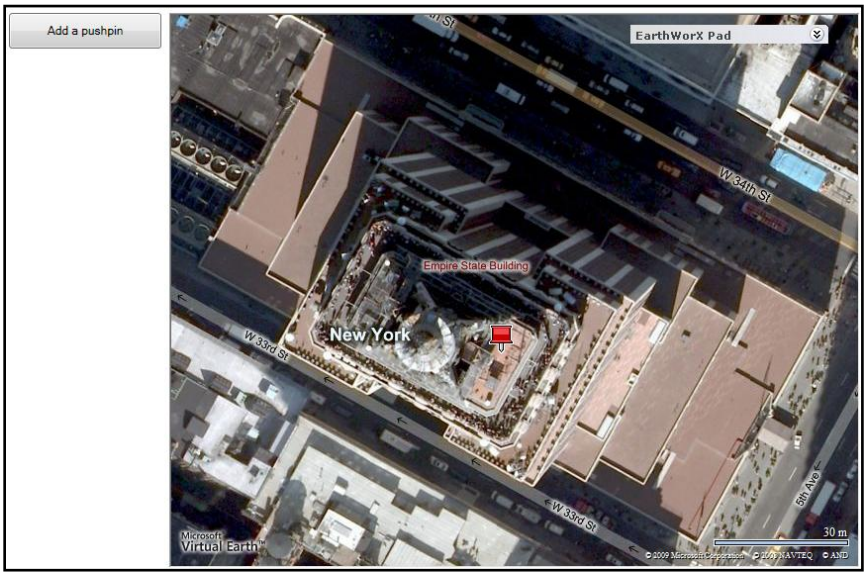


Figure 5.19 - A Detailed (Aerial) View of our Pushpin Location

To resume our example, three lines of code were enough to add a new object on the map. In a real world application, you may probably want to read your objects' information from an external data source (file, database, web services, etc.) and create a loop to add all of them all at once. This can speed up the configuration process greatly.

5.6.3 Navigating between Different Locations on the Map

EarthWorX allows you to browse the map freely. However, sometimes it may be extremely useful to have some navigation button to quickly jump to a place, zoom in or out, and switch the map mode. You can easily use EarthWorX API to do that. In this section, we will present a simplified version of this in order to introduce the concept, and in next section, we will make some improvement.

Before proceeding, we need to define a new concept, the "view". In EarthWorX a view is a set of information used to define "what you see" on the map. This usually consists of the geographical position of the center of the map, the current zoom level, and the map style (i.e. Road, Aerial, and Bird's eye).

A view in EarthWorX is defined using the `Ico.Ewx.Objects.View` object. You can programmatically move the map by scripting, using the "SetView" method of the map object. This method takes the view as its parameter and will center the map to the specified position, adjust the zoom and map style to match with the ones defined in the view, if needed.

We want to create an example, where we define three different buttons that will bring us to three different locations when clicked. To achieve this, we will first define a utility function that we will reuse for each button. We will call this function SetView.

```
function SetView(lat : double, lon : double, zoom : int,
mapStyle : Ico.Ewx.Objects.MapStyles)
{
      var map : Ico.Ewx.Objects.Map =
      GetMap("EwxControl1");

      map.SetView(new Ico.Ewx.Objects.View("", lat, lon,
      zoom, mapStyle));
}
```

This function takes four parameters: two to define the geographical position, the zoom, and then the map style. The map style is defined in an enumeration called `Ico.Ewx.Objects.MapStyles`.

The function gets a reference to the map, and then using the map's SetView method passing it a view, which gets created on-the-fly from the supplied parameters.

We are going to use this function for the buttons we intend to create. We want to allow our users to jump to New York City, San Francisco, and to an overview of the United Stated quickly. For this reason, we create three buttons and three functions associated to them.

```
function GoToNewYork(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      SetView(40.73, -74, 10,
      Ico.Ewx.Objects.MapStyles.Hybrid);
}

function GoToSanFrancisco(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      SetView(37.77, -122.43, 12,
      Ico.Ewx.Objects.MapStyles.Aerial);
}

function GoToUsa(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      SetView(40, -97, 4, Ico.Ewx.Objects.MapStyles.Road);
}
```

These three functions are almost the same, except for the parameters they pass to the SetView function. Do not confuse this function with the map object SetView method. For this reason, you may want to rename the function you define to SetUserView.

Once we have completed the creation of our buttons and functions we can save the display and go back to runtime.

Right now, after adding the Pushpin object to the map, we can click the "New York City" button to display the location quickly in the correct zoom level, without having to navigate manually through the map.

Figure 5.20 - A Hybrid View over New York City

Clicking the San Francisco button will instantly move the map to the West Coast city, and as you can see, the map style has changed from Hybrid to Aerial (a satellite view with no labels on it.)

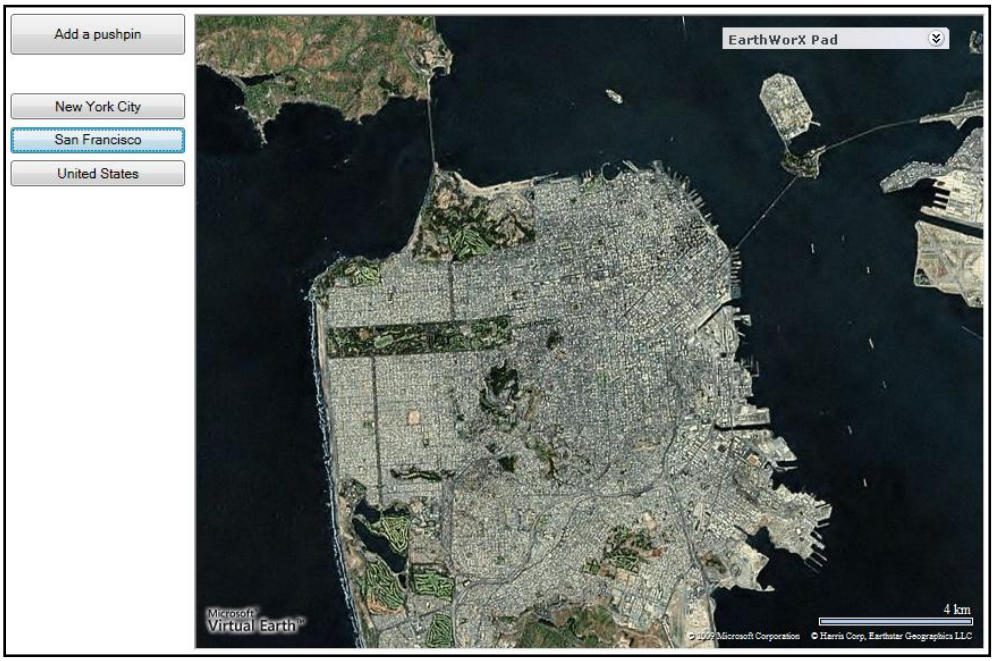


Figure 5.21 - An Aerial View of San Francisco

The “United States” button will take the view back to an overview of the country, also changing the style to “Road”.

Figure 5.22 - A Road View of the United States

As you can see, we have defined a navigation system to let the user quickly jump to their places of interests. If you have several geographical locations where your assets are located, you may find it convenient to define a similar system.

5.6.4 Improving the Navigation System

The navigation system we presented in the previous section is an extremely valuable tool to speed up navigation with the map. However, it has a drawback: you need to define a function for each view. This may be ok if you have a fixed set of views that you always work with, but does not scale well if you need to maintain the display, adding and removing views.

In this section, we present a useful concept that allows creating an enhanced navigation system. When you write a script function, you have a very useful property of your Pick Action, called “CustomData”. This property can be anything you like and can be accessed from the script function associated to this pick action.

We can take advantage of this to pass some parameters to our function. Since this property can be a simple string, we will write all of our parameters in a csv-like

way, which is a comma separated list of values. Consider, for example, Figure 5.23.

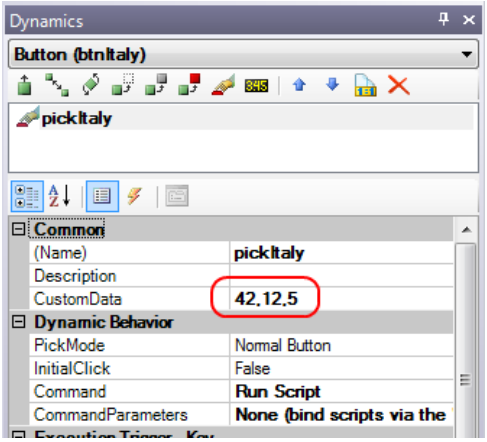


Figure 5.23 - Specifying Custom Parameters

Here, we have a button we want to use to load Italy into our map quickly. First, we define our custom data as the Latitude and Longitude coordinates for Italy and the zoom level. To keep the example simple, we ignore the map style for now, but you can easily extend the example to consider that extra parameter as well.

In the previous section, we have defined the "SetView" function, which was taking four different parameters. We will now define an enhanced version of that function, which will read its parameters from the CustomData property instead.

```
function SetViewEnhanced(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var params : String[] = sender.CustomData.Split(",");
      var lat = params[0];
      var lon = params[1];
      var zoom = params[2];
      SetView(lat, lon, zoom);
}
```

As you can see, we need to handle our raw data coming from the CustomData property. The Split method helps us to obtain an array of String values, which are our parameters. In this simple example, we can ignore the data types; the system will handle that for us.

Once the function has been defined, all we need to do is to assign it to the button we created, as in Figure 5.24.

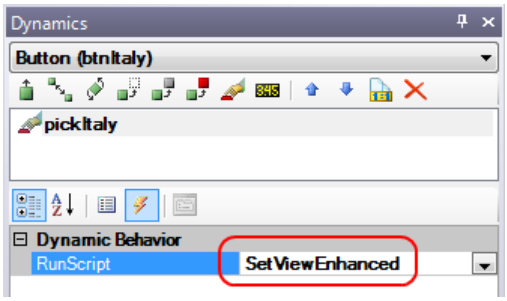


Figure 5.24 - Assigning the New Function to the Button's Pick Action

Using this approach, we no longer need to define a function for each button. Suppose we want to create three buttons to navigate to Italy, Germany, and Ireland, we can now just duplicate the "Italy" button to create another two, and simple alter those buttons' CustomData properties, as show in the next figure.

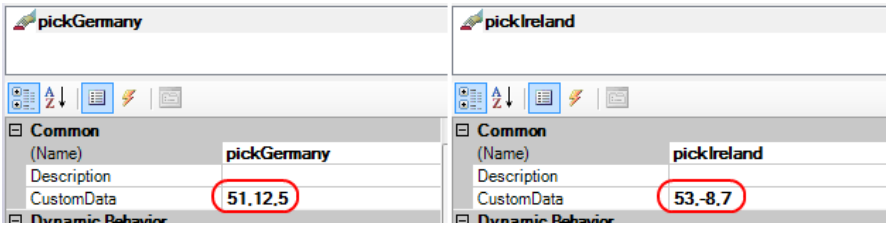


Figure 5.25 - Setting the CustomData Properties for Other Buttons

Going into runtime mode and clicking on Italy will move the map there as shown in Figure 5.26.

Figure 5.26 - A View of Italy

When clicking on Germany, the map will move there. If you were focused on Italy before, you probably noticed a smooth animation to Germany, this automatically happens when you switch to a view that is not too far away from your current location.

Figure 5.27 - A View of Germany

And finally, we click the Ireland button, which also zooms in the view a bit.

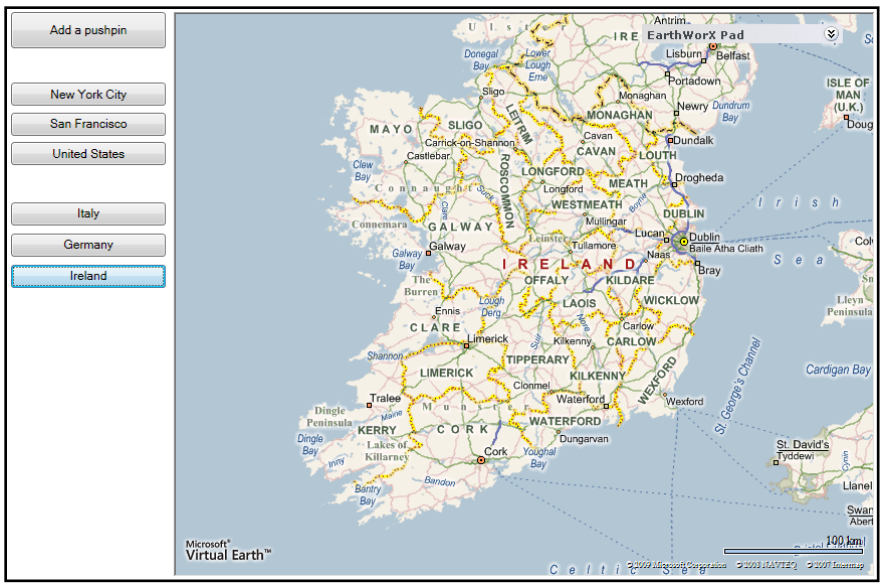


Figure 5.28 - A Closer View of Ireland

This approach reduced the number of functions we need to define. It is important to note that this is not specific to EarthWorX. You can use the CustomData property anywhere you need to write a script. It can be very helpful in reducing the amount of code that you need to maintain.

5.7 Conclusions

We have seen an overview of the APIs exposed by GENESIS64 Controls and how to use them from inside GraphWorX64. It is important to note that all of these tasks can also be performed inside an external project, written in your favorite .NET language. It is also important to note that if you plan to write your own .NET Controls to be used inside GraphWorX64, then you can also define and use your own controls' API by scripting in the same way.

Regardless if you are using GENESIS64 Controls, Windows Controls, or your own custom ones, before you start scripting, you should always make sure that you have included the necessary references. Each of the examples we revealed in this chapter requires adding extra references. Without properly adding these, your scripts will error.

Aside from understanding the different API available in the different controls, it is also imperative that you understand the scripting environment and know the tools that can help reduce your efforts with scripting. For example, CustomData is a good way to reduce the amount of code that you need to write.

Lab: Scripting with ICONICS' Controls *5*

5.1 Lab Overview

IcTonics, being a global company, would like a map to show off the different locations of their offices and plants. This map should be able to zoom into a particular location with a press of the button.

With the information we learned from this chapter, it should be simple to give icTonics the functionality they need.

Estimated time to complete this lab: 20 minutes

5.1.1 Task Checklist

In this lab, you will complete the following tasks:

- Limit acknowledging alarms to only available through buttons you create
- Create a script to acknowledge selected alarms
- Create a script to acknowledge all alarms at once
- Create a display that allows zooming into different locations

5.1.2 Support Files

You will need the following files to complete this lab:

- icTonicsTankMonitor.gdfx
- icTonicsTemplate.tdfx

5.2 EarthWorX Viewer

IcTonics is an international company with plants and offices in different locations of the world. They would like to create a display that allows them to view information about a particular plant quickly.

1. Create a new GraphWorX64 display based on the icTonicsTemplate.tdfx and save the display as "icTonicsWorldMap".
2. Add the EarthWorX viewer into this display by going to the Controls Tab of the ribbon and clicking the EarthWorX64 Viewer.
3. The size of the EarthWorX viewer should be 900 x 675. Move the control so that the top of it is just below the banner.
4. Create a button and put it to the right of the EarthWorX viewer.
5. In the CustomData field of the pick action, enter "0,0,1".
6. Change the command of the button to "Run Script" and create a script called "icTonics_SetView".
7. Once you are in the script editor, click the Reference button.
8. In the References dialog, scroll to an empty line and click the … button in the Assembly Codebase section.
9. Browse to \Program Files\ICONICS\GENESIS64\Components and look for the EwxApi.dll. Select this file and click open to add it as a reference.
10. Write the script in here that takes the CustomData as the input to your location and zoom level. The specific code was discussed in the chapter and will not be shown here again; but you can use the pseudo code below as a guide.

```
function icTonics_SetView(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      //get controls from display
      var wfControl
      var ewxViewer
      var map

      //set the view using CustomData
      map.SetView(CustomerData[1], CustomerData[2],
      CustomerData[3]…);
}
```

11. Once you are done writing the script, you can go into runtime to test it. If it works as you expect it to, duplicate the button three times, and change the properties as indicated in Table L5.1.

Table L5.1 - CustomData

Button Text	Custom Data
Foxboro	42.042, -71.235, 14
Plzen	49.746, 13.373, 11
ShenZhen	22.544, 114.101, 16

12. Once you have finished configuring your buttons, you can go into runtime again and click the different buttons to see that it does bring you to the different locations. Your finished display may look similar to Figure L5.1.

Figure L5.1 – icTonicsWorldMap

5.3 Advanced Lab

IcTonics would like to add a few buttons on the icTonicsTankMonitor.gdfx to let the operators acknowledge alarms. They would also like a special button that only those with permission can execute the script to acknowledge all alarms at the same time. To do this, we will first need to disable the double-click to acknowledge alarms on the viewer and before adding our scripts.

1. Open the icTonicsTankMonitor display and open the configuration for the alarm viewer in it.

2. Select the AlarmWorX64Viewer on the left-hand pane if it's not already selected.

3. In the Appearance tab on the right-hand pane, uncheck the "Show Toolbar" option. This will disable the runtime toolbar where the user can use it to acknowledge alarms. Your Appearance tab should look similar to Figure L5.2.

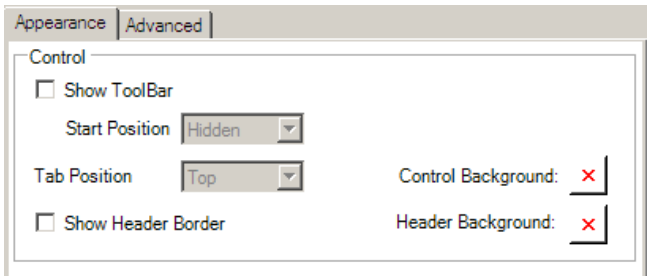


Figure L5.2 - Alarm Viewer Appearance Tab

4. Select the Grid on the left-hand pane and select the Behavior tab on the right-hand pane.

5. Uncheck the Enable Popup Menu option. This will limit the user from trying to acknowledge an alarm using the popup menu.

6. In the Acknowledgement section, make sure that the "Enable Double Click" option is unchecked. Your Behavior's tab should look similar to Figure L5.3 when you are done.

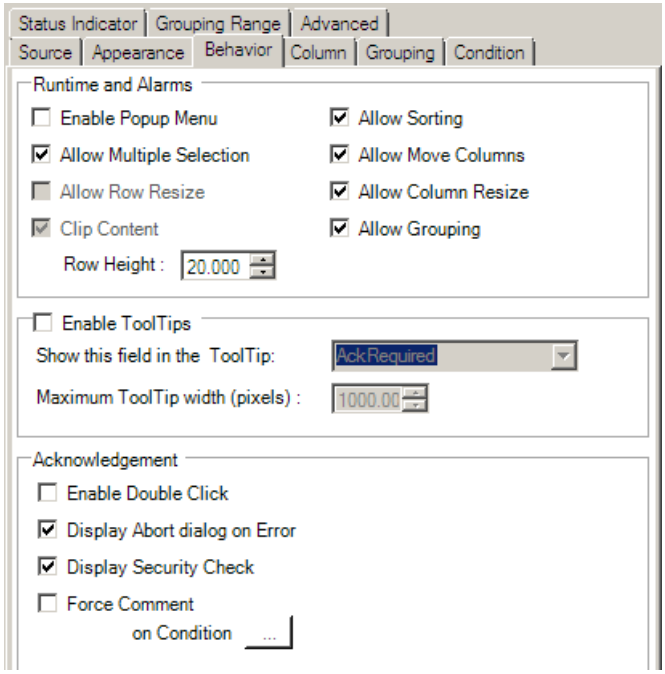


Figure L5.3 - Alarm Grid Behavior Tab

7. Click on "OK" to apply the changes and exit configuration for the control.

8. You can go into runtime to test that you cannot double-click to acknowledge an alarm or bring up the right-click menu to do it.

9. Create two buttons to the right of the alarm viewer. Both should execute a script. The first button will allow acknowledgement of the selected alarms and the second will allow acknowledgment of all alarms.

10. For the first button, give it the text "Ack Selected" and associate a script called "icTonics_AckSelected" to the button.

11. When the button is clicked, it should acknowledge those alarms that are selected. Below is the pseudo code for it:

```
function icTonics_AckSelected(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      //declare and set variables
      var viewerObject = AlarmViewer on Display;
      var view = default view in viewerObject;

      view.AcknowledgeSelected;
}
```

12. Once you have the script written, go into runtime to test it.

13. Now to configure the second button, remember that we only want administrators to be able to acknowledge all alarms at once. So must add security to this button.

14. In the security configuration, open the properties for the Administrator user. In the Custom tab, enter SecuredItems into the allow section.

15. Make sure that the other users, namely Operator and Operator2, do not have permissions to access Custom words "SecuredItems". You can try testing it with the test string for the particular user.

16. In the GraphWorX64 display, select the button that you want to use for acknowledging all alarms, and give it the text "Ack All".

17. In the Dynamics Property of the button (the pick action that will run your script), set the CustomerData to "SecuredItems".

18. Also, name the Pick Action "AckAll" so that you can refer to it in your code.

19. Create a function called "icTonics_GlobalAck" and associate to the button. In this function, you should be able to acknowledge all alarms by pressing it. The

code should be similar to what you wrote for acknowledging the selected alarms with two differences: one that checks for security and the other to acknowledge all current alarms. Below is the pseudo code

```
function AckAll(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      //get the button and get the custom data from it
      var button = DynamicObject("AckALL");

      if
      (ThisWindow.IsCustomSecurityAllowed(button.CustomData
      )){
            acknowledge the all the alarms
      }else you don't have permission
}
```

Note: The IsCustomSecurityAllowed method is only available in version 10.5. If you are not using this version, you can try getting the security by using a process point and the special security syntax or put the button on a secured layer to achieve the same result.

20. When you are done writing the script, go into runtime and test it out. Without logging in as the administrator user, notice that you cannot execute the code behind the "Ack All" button.

Scripting with Controls Reviewed

5.1 Objectives Review

After completing this chapter, you should know how to:

- Obtain a reference to a GENESIS64 Control to work with it
- In the Alarm Viewer
  - Create a new tab
  - Toggle the Groping Feature
  - Perform a Global Acknowledge
- In the Trend Viewer
  - Add a new pen
  - Remove the last pen
  - Toggle the legend's visibility
  - Swap the trend axis
- In the EarthWorX Control
  - Create a pushpin and place it on the map
  - Navigate between different locations on the map
  - Use CustomeData to better manage scripts

5.2 Summary

Throughout this chapter, we have shown you how to work with GENESIS64 Controls. In order to work with any types of controls using scripting, whether it is one provided by ICONICS, Microsoft Windows, or your own customer one, you must first get a reference to the control on the display. The steps involved usually include naming the control on the display, declaring a variable in the script of the type of control, and then finally get the object from the display and set it to the variable. Once you have obtained the control in your script, you should be able to access its API and make it perform the function you wish.

Regardless of the type of controls you wish to use, before you start scripting with it, you should always add the reference – the file that contains the API methods – to your scripting project first. Without this, you may encounter errors when use the different APIs for the control.

We presented sample scripts for the three commonly used GENESIS64 controls: AlarmWorX64 Viewer, TrendWorX64 Viewer, and EarthWorX. While AlarmWorX64 Viewer and TrendWorX64 Viewer are similar, EarthWorX viewer is different in that it is a WPF control. Furthermore, AlarmWorX64 and TrendWorX64 viewers are organized with tabs and views, while EarthWorX uses a single map object to display the information.

As the final point in the chapter, we presented a way to script with GraphWorX64 CustomData for an object to help reduce the amount of code and script. The way this works is by passing different parameters using the CustomData field into the script, in turn, you only need to write one function, as opposed to many, giving it different parameters to perform similar task with different results.

5.3 Terms

Alarm Viewer Chart	Alarm Viewer Grid	Alarm Grouping
Alarm Viewer Tab	Alarm Views	CustomData
DLL	EarthWorX	EarthWorX Map
EarthWorX View	Pushpin	References
Trend Viewer Series	Windows Form	WPF

5.4 Test Your Knowledge

In this chapter, we discussed three different controls and some of the API available in each. You should be able to answer the questions below after learning about the topics in this chapter.

1. Which is <u>NOT</u> true about the EarthWorX Viewer

    a. You can add shapes to it using API
    b. You can add views to it using API
    c. You can add different tabs in the control using API
    d. It is a WPF control

2. True or False: You must add the reference for a particular control in order to use the available APIs.

3. What is CustomData?

    a. A property for GraphWorX64 objects
    b. Parameter field used to pass data from an object to associated scripts
    c. Data entered manually by a user during runtime
    d. All of the above
    e. Both A and B
    f. Both A and C
    g. Both B and C
    h. None of the above

4. Which Controls does GraphWorX64 support in addition to ICONICS controls?

    a. WPF Controls
    b. Windows Form
    c. 64-bit ActiveX
    d. All of the Above
    e. Only A and B
    f. Only A and C
    g. Only B and C
    h. None of the above

5. True or False: You can declare helper functions not tied to a button to reuse general code or codes that you need repeatedly.

6. You can use automation to perform which of the following alarm acknowledgment?

    a. Acknowledge all alarms
    b. Acknowledge only filtered alarms
    c. Acknowledge selected alarms
    d. All of the above
    e. Only A and B
    f. Only A and C
    g. Only B and C
    h. None of the above

Glossary

Alarm Viewer Chart is an object within the AlarmWorX64 Viewer. It resides in the *Alarm Viewer Tab* and displays a chart to shows alarm summary information.

Alarm Viewer Grid is an object within the AlarmWorX64 Viewer. It resides in the *Alarm Viewer Tab* and displays the current alarms in a table-like format.

Alarm Grouping is a feature in the AlarmWorX64 Viewer. It allows you to group alarms with similar attributes. You can give one or more grouping criteria to organize the displayed alarms in a manner that is more meaningful and useful to you.

Alarm Viewer Tab is an object within the AlarmWorX64 Viewer. It allows you to organize your alarms into a different "sheet" or tab in the viewer.

Alarm Views are the combination of *Alarm Viewer Chart* and *Alarm Viewer Grid*, which displays alarm information.

Analog Animation controls the animation of a dynamic based on the value of the data source. You can also call it a data-driven animation as the animate sequence and length are based on the current value of the data source.

Color Dynamic is one of the many dynamics available in GraphWorX64. It allows you to change the color of an object.

CustomData is a property of any GraphWorX64 object. Using this property, you can pass user data into your script function, enhancing code management.

Discrete Animation allows a timed-based animation. You can set the animation length by configuring the duration property in milliseconds. The start and end of the animation depends on the ActiveWhen condition and the value of the data source.

DLL stands for Dynamic Link Library. These files usually contain the automation methods you can use with scripting. If any is required, you should add it as a *reference* to your scripting project.

Dynamics are "animations" that you can add to objects on a GraphWorX64 display. It usually contains a condition in which objects are manipulated to follow a set of rules based on a data source.

EarthWorX is a component of GENESIS64. It uses the Microsoft's Virtual Earth technology to bring you maps, onto which you can add shapes such as *Pushpins* to show location specific data.

EarthWorX Map is an object within the *EarthWorX Viewer*, which you must obtain in order to work with it using automation methods.

EarthWorX Viewer is the control that allows us to display maps in your project.

Hide Dynamic is one of the many dynamics available in GraphWorX64. It allows you to hide or show an object.

IntelliSense is a tool in the GraphWorX64 scripting editor that shows you the different functions and properties available along with the necessary parameters.

JScript.NET is the scripting language that GraphWorX64 uses. It belongs in the same .NET family of programming languages such as C#.

Locations Dynamic is one of the many dynamics available in GraphWorX64. It allows you to change an object's location on the display creating the illusion that it is moving.

Pushpin is one of the shapes you can add onto the *EarthWorX Map* to display location specific data.

Range Selectors are dynamics that you can apply to a group of object. It is considered as a *Discrete Animation* in which a set of objects are cycled through at runtime based on the configured duration property.

References are files that you must associate to your scripting project in order to the automation methods included for your control. You can consider these "include" files.

Rotation Dynamic is one of the many dynamics available in GraphWorX64. It allows you to rotate an object around a center you configure.

Size Dynamic is one of the many dynamics available in GraphWorX64. It allows you to shrink and grow an object in runtime.

Smart Properties are used by *Smart Symbols* to set one or more properties of child or descendant objects.

Smart Symbol is an enhanced version of object groups with custom properties.

State Selectors are dynamics that you can apply to a group of object. It is considered as an Analog Animation where the value of the data source controls which object in the group to display. You can compare this to having *Hide Dynamics* on different objects controlled by the same data source.

Symbols Library is our collection of Symbols or graphics with predefined properties, which can be used in GraphWorX64 to minimize application development time.

ThisConfiguration is a variable most comparable to ThisDisplay in GENESIS32. You can access functions to manipulate objects such as process points and shapes using this variable.

ThisDocument is a variable used in the GraphWorX64 scripting. You can access the runtime functionality including options found in the properties tab in configuration using this variable.

ThisWindow is a variable used in the GraphWorX64 scripting. You can access properties of the GraphWorX64 window such as dimensions of the viewing area or converting point coordinates using this variable.

Trend Viewer Series resides in a trend view, which may contain one or more pens to draw values in runtime.

Windows From is a control that you can use within GraphWorX64. The DataGridView control falls in this category.

WPF stands for Windows Presentation Foundation, which is crucial to the graphic display in GENESIS64. Using *WPF*, you can create controls like our *EarthWorX Viewer*.

Answers

Questions / Chapters	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Chapter 1	True	D	B	A	G					
Chapter2	True	False	True	False	D	D				
Chapter3	C	A	False	D	A	True	False			
Chapter4	E	False	D	True	True					
Chapter5	C	True	E	D	True	D				

Complete Scripts for Labs *C*

C.1 Lab 3: Basic Scripting

C.1.1 Code for Step 5

```
function icTonics_SimulateData(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      //Declare the variables
      var syrup : GwxDynamic = ThisConfiguration.
      GetDynamicObjectByName("Open");
      var water : GwxDynamic = ThisConfiguration.
      GetDynamicObjectByName("Open1");
      var tankLevel : GwxDynamic = ThisConfiguration.
      GetDynamicObjectByName("TankLevel");
      var temp : Number = 0;

      //Add the new values and give it to tankLevel
      if (syrup.GetValueOfPrimaryDataSource()) temp = temp
      + 1;
      if (water.GetValueOfPrimaryDataSource()) temp = temp
      + 1;

      tankLevel.WriteValueToPrimaryDataSource(temp +
      tankLevel.GetValueOfPrimaryDataSource());
}
```

C.1.2 Code for Step 9

```
function icTonics_SimulateData(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
      var syrup : GwxDynamic = ThisConfiguration.
      GetDynamicObjectByName("Open");
      var water : GwxDynamic = ThisConfiguration.
      GetDynamicObjectByName("Open1");
      var tankLevel : GwxDynamic = ThisConfiguration.
      GetDynamicObjectByName("TankLevel");
      var temp : Number = 0;

      if (tankLevel.GetValueOfPrimaryDataSource() >= 100){
            syrup.WriteValueToPrimaryDataSource(false);
```

```
                water.WriteValueToPrimaryDataSource(false);
        }else {
                if (syrup.GetValueOfPrimaryDataSource()) temp =
                temp + 1;
                if (water.GetValueOfPrimaryDataSource()) temp =
                temp + 1;
                tankLevel.WriteValueToPrimaryDataSource(temp +
                tankLevel.GetValueOfPrimaryDataSource() );
        }
}
```

C.2 Lab 4: Viewing Trend Data in a Table

C.2.1 Code for Step 3

```
function icTonics_RefreshTrendData(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
        var strConnection : String = "Data
        Source=(local);Initial
        Catalog=icTonicsTrendData;Integrated Security=SSPI";
        var connection : SqlConnection = new
        SqlConnection(strConnection);
        var strCommand;
        strCommand = icTonics_commandString();

        var command : SqlCommand = new SqlCommand(strCommand,
        connection);
        var dataAdapter : SqlDataAdapter = new
        SqlDataAdapter(command);
        var dataTable : DataTable = new DataTable();

        dataAdapter.Fill(dataTable);

        var wfCtrl : GwxWindowsFormsControl =
        GwxWindowsFormsControl(ThisConfiguration.GetObjectByN
        ame("DataGridView1"));
        var grid : System.Windows.Forms.DataGridView =
        System.Windows.Forms.DataGridView(wfCtrl.Control);

        grid.DataSource = dataTable;
}

function icTonics_commandString()
{
        var strCommand : String;
        strCommand = "SELECT * From TankLevels_1";
        return strCommand;
}
```

C.2.2 Code for Step 5

The simplest way to do this is to create a view for the database in SQL. Assuming you know how to create this view, the code should be as below. Since the only thing we need to change is the SQL command string we will only show the code for this part. The code assumes that you have a view called *TWXLogData*.

```
function icTonics_commandString()
{
      var strCommand : String;
      strCommand = "SELECT * FROM TWXLogData Order by
      Logging_Name asc, Sample_TDate_1 asc";
      return strCommand;
}
```

If you want to try creating your own view, you can use the following code, changing the *ViewName*, table names TankLevels_Tags and TankLevels_1 accordingly.

```
CREATE VIEW ViewName AS
SELECT TankLevels_Tags.Logging_Name,
TankLevels_1.Sample_TDate_1, TankLevels_1.Sample_MSec_1,
TankLevels_1.Sample_Value_1 FROM TankLevels_1,
TankLevels_Tags WHERE TankLevels_1.Sample_TDate_1 IS NOT
NULL and TankLevels_1.Signal_Index =
TankLevels_Tags.Tag_Index UNION
SELECT TankLevels_Tags.Logging_Name,
TankLevels_1.Sample_TDate_2, TankLevels_1.Sample_MSec_2,
TankLevels_1.Sample_Value_2 FROM TankLevels_1,
TankLevels_Tags WHERE TankLevels_1.Sample_TDate_2 IS NOT
NULL and TankLevels_1.Signal_Index =
TankLevels_Tags.Tag_Index UNION
SELECT TankLevels_Tags.Logging_Name,
TankLevels_1.Sample_TDate_3, TankLevels_1.Sample_MSec_3,
TankLevels_1.Sample_Value_3 FROM TankLevels_1,
TankLevels_Tags WHERE TankLevels_1.Sample_TDate_3 IS NOT
NULL and TankLevels_1.Signal_Index =
TankLevels_Tags.Tag_Index UNION
……
……
……
SELECT TankLevels_Tags.Logging_Name,
TankLevels_1.Sample_TDate_36, TankLevels_1.Sample_MSec_36,
TankLevels_1.Sample_Value_36 FROM TankLevels_1,
TankLevels_Tags WHERE TankLevels_1.Sample_TDate_36 IS NOT
NULL and TankLevels_1.Signal_Index =
TankLevels_Tags.Tag_Index
```

C.2.3 Code for Step 6

Again for this step, everything in icTonics_RefreshTrendData() is unchanged, the only thing we change is the icTonics_commandString() function. Your code should look similar to:

```
function icTonics_commandString()
{
      var strCommand : String;
      var twxTag : GwxDynamic =
      ThisConfiguration.GetDynamicObjectByName("twxTag");
      if (twxTag.GetValueOfPrimaryDataSource() == ""){
            strCommand = "SELECT * FROM TWXLogData Order by
            Logging_Name asc, Sample_TDate_1 asc,
            Sample_MSec_1 asc";
      }else{
            strCommand = "SELECT * FROM TWXLogData WHERE
            Logging_Name = '" +
            twxTag.GetValueOfPrimaryDataSource() + "' Order
            by Logging_Name asc, Sample_TDate_1 asc,
            Sample_MSec_1 asc";
      }
      return strCommand;
}
```

C.3 Lab 5: Scripting with ICONICS' Controls

C.3.1 EarthWorX64 Viewer: Code for Step 11

```
function icTonics_SetView(sender : System.Object, cmdArgs :
Ico.Gwx.CommandExecutionEventArgs)
{
      var wfControl : Ico.Gwx.GwxWindowsFormsControl;
      var ewxViewer : Ico.Ewx.EwxControl;

      wfControl =
      Ico.Gwx.GwxWindowsFormsControl(ThisConfiguration.GetO
      bjectByName("EwxControl"));
      ewxViewer = Ico.Ewx.EwxControl(wfControl.Control);

      var map : Ico.Ewx.Objects.Map;
      map = ewxViewer.GetMap();
      var param : String[] = sender.CustomData.Split(",");
      map.SetView(new Ico.Ewx.Objects.View("", param[0],
      param[1], param[2], Ico.Ewx.Objects.MapStyles.Road));
}
```

C.3.2 Advanced Lab: Code for Step 11

```
function icTonics_AckSelected(sender : System.Object,
cmdArgs : Ico.Gwx.CommandExecutionEventArgs)
{
	var myView : Ico.Awx.AwxViewControl;
	myView =
	ThisConfiguration.GetObjectByName("AlarmWorX64Viewer"
	).ToDependencyObject();

	var view : Ico.Awx.DataGrid.AwxGridView;
	view =
	Ico.Awx.DataGrid.AwxGridView(myViewer.Tabs[0].Views[0
	]);
	view.Acknowledge(Ico.Awx.DataGrid.AckType.Selection);
}
```

C.3.3 Advanced Lab: Code for Step 19

```
function icTonics_GlobalAck(sender : System.Object, cmdArgs
: Ico.Gwx.CommandExecutionEventArgs)
{
	var myButton : GwxDynamic
	myButton =
	ThisConfiguration.GetDynamicObjectByName("AckAll")

	if
	(ThisWindow.IsCustomSecurityAllowed(myButton.CustomDa
	ta)){
		var myView : Ico.Awx.AwxViewControl;
		myView =
		ThisConfiguration.GetObjectByName("AlarmWorX64V
		iewer").ToDependencyObject();

		var view : Ico.Awx.DataGrid.AwxGridView;
		view =
		Ico.Awx.DataGrid.AwxGridView(myViewer.Tabs[0].V
		iews[0]);
		view.Acknowledge(Ico.Awx.DataGrid.AckType.All);
	}else
		MessageBox.Show("Sorry, you do not have
		permission to perform this action.");
}
```

If you are using a version prior to 10.5, you can try the following code instead:

```
function icTonics_GlobalAck(sender : System.Object, cmdArgs
: Ico.Gwx.CommandExecutionEventArgs)
{
	var securityPoint : GwxDynamic
	securityPoint =
	ThisConfiguration.GetDynamicObjectByName("CustomSecur
	ity");
```

```
        if (securityPoint.GetValueOfPrimaryDataSource() !=
        0){
              var myView : Ico.Awx.AwxViewControl;
              myView =
              ThisConfiguration.GetObjectByName("AlarmWorX64V
              iewer").ToDependencyObject();

              var view : Ico.Awx.DataGrid.AwxGridView;
              view =
              Ico.Awx.DataGrid.AwxGridView(myViewer.Tabs[0].V
              iews[0]);
              view.Acknowledge(Ico.Awx.DataGrid.AckType.All);
        }else
              MessageBox.Show("Sorry, you do not have
              permission to perform this action.");
}
```

NOTE: In order for the above code to work, you must have a process point named "CustomSecurity". The data source for this tag should be "?Custom:SecuredItems". For more information on this special syntax, please refer to the security chapter.

ICONICS

OPC Automation at your fingertips

100 Foxborough Blvd., Foxborough, MA 02035
508.543.8600 • FAX 508.543.1503 • BBS 508.543.3881
www.iconics.com • E-mail info@iconics.com

© 2012 ICONICS, Inc. All Rights Reserved.
Specifications subject to change without notice.