Using Multiple Monitor Configurations with InduSoft Web Studio

Abstract

The reasons for using multiple monitors with InduSoft Web Studio can vary widely due to the design, environment, and the intentions of the application that is being built. As a result, many differing and unique configurations and solutions may be required that are not normally considered during application development that will use a single display or Thin Client. This Tech Note discusses most of the issues and configurations that might be encountered when using multiple monitors. Comments for improving this Tech Note are welcomed. Please send your Tech Note feedback to info@indusoft.com. Additionally, suggestions for product improvement or additional features can be sent to requirements@indusoft.com.

Applicable IWS Versions or Requirements

Category	Item	
Software	IWS Version:	7.1 and later
	Service Pack:	SP2 and later
	Windows Version:	Win 7, 8, 2008/12 Server, W7SE, and later
	Web Thin Client:	Yes
Equipment	Panel Manufacturer:	N/A
	Panel Model:	N/A
	Other Hardware:	External Monitor or Monitor Array
	Comm. Driver:	All
	Controller (e.g.: PLC)	All
	Application Language:	N/A
Software Demo Application	Discussed, downloadable in Sample Applications: MultiMonitorProjects.zip	
Date Released	March 3, 2014	
Author	RHC	

General Introduction and Assumptions

Since many differing development and runtime environments exist, it will not be possible to explore every possible configuration individually within this document. Generally speaking however, discussions can be made about specific configurations which will apply to a number of similar types of environments. In other words, suggestions for one type of environment may be perfectly acceptable to use in a different one. It is advisable therefore to thoroughly test the InduSoft Web Studio project several times within the expected runtime environment during the development phase to make sure that the screens and objects display properly and as expected, and that your runtime machine will have sufficient resources to encounter any situation that the new project and operating environment may provide.

The suggestions and discussions within this Tech Note can be applied to all currently supported versions of InduSoft Web Studio running in supported versions of Microsoft Windows, except as noted within the discussions. Additionally, certain viewer configurations may consume many resources on the runtime machine(s), so ongoing testing of the project on the actual target machine(s) or a valid simulator, from time-to-time during the development phase, becomes a necessity in order to make proper allowances within the project or the runtime machine itself (e.g., upgrading resources or replacing it with a faster machine).

Contents

Section 1: Background Information and Review

InduSoft Internal Architecture Review with Focus on the Viewer Module

When designing applications that will run on more than one monitor or that may use one or more Secure Viewers (***viewer.exe process***), a review of the InduSoft Web Studio architecture will be helpful. The basic internal architecture is depicted in ***Figure 1***.

Before proceeding, it will be helpful for the reader to open the ***Help Document***, and read the entire first chapter called "***Introduction***". Pay particular attention to the sections, "***Internal Structure and Data Flow"***, "***Executing and Switching Modules***, and "***Executing and Switching the Background Task***", with the "***Internal Structure and Data Flow***" being the most important section. Additionally, it is important to understand the distinction between ***Local Scope*** and ***Server Scope*** tags.

At the heart of InduSoft Web Studio is the ***Tags Database***. Every piece of functionality for the product relies on the ***Tags Database*** and the state of any tag that is currently being serviced. All functionality of InduSoft Web Studio is external to the Tags Database and each module of the product runs in a separate process thread apart from the other modules as depicted in ***Figure 1***; therefore it is not possible to control the operation of each piece of functionality except by (1) configuration within the ***Development Environment***, (2) the available ***System Functions***, and (3) the state of the ***Server Scope*** tags.

Following this scheme, it is possible to use several ***viewer.exe*** module processes simultaneously, which are independent from each other. In fact, this is exactly what occurs when a Thin Client is used. The Plug-In ***ISSymbol.ocx*** (a Graphics Module Plug-In for Internet Explorer) is loaded into an ***Internet Explorer*** when a ***TCP/IP*** connection is made to the InduSoft Web Studio Runtime by browsing to an HTML rendered project screen or screen group; and a new Thin Client Viewer is thereby opened (See ***Figure 1***). For purposes of this discussion however, we will only consider the (Secure Viewer) Graphics Modules (***viewer.exe***) which effectively contains an embedded ISSymbol object within it.

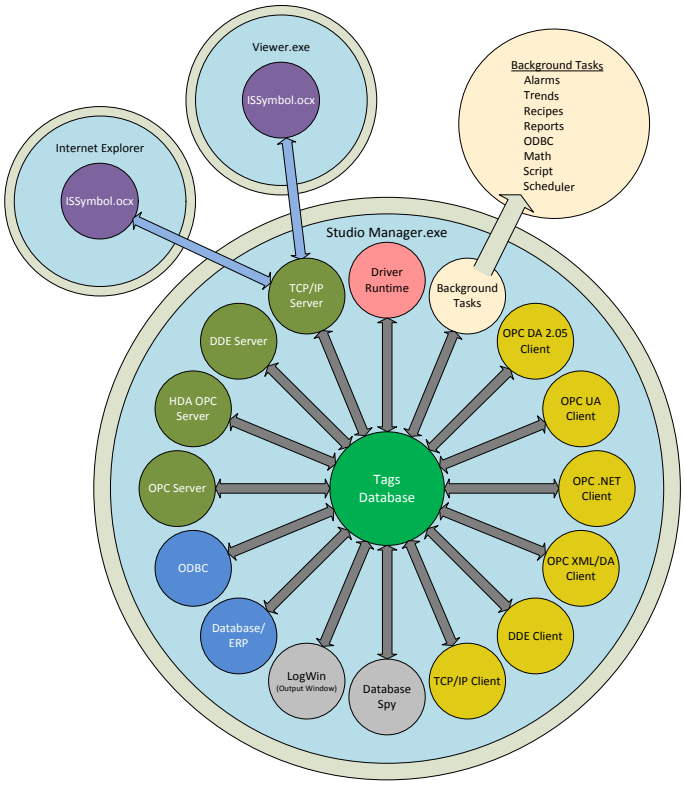

Figure 1: InduSoft Web Studio internal architecture

It is important to note that the modules such as the ***Drivers***, ***Database/ERP***, and ***Background Task Modules,*** for instance, will not be affected by ***Local Scope Tags*** used on each of the instantiated graphics modules (***viewer.exe process***). Reviewing the layout of the IWS architecture in ***Figure 1*** makes it easy to understand why this is the case.

Therefore, the only scripts that will be directly available **locally** to each graphics module are the:

1. **Graphics Scripts**, providing the ability to affect local and server tags during ***Graphics_OnStart()***, ***Graphics_WhileRunning()***, and ***Graphics_OnEnd()*** modes of each of the graphics modules (***viewer.exe processes***) that are instantiated and destroyed.
2. **Screen Scripts**, providing the ability to affect local and server tags during ***Screen_OnOpen()***, ***Screen_WhileOpen()***, and ***Screen_OnClose()*** modes of each of the screens being displayed on any particular graphics module (***viewer.exe processes***).
3. **Object Scripts**, such as ***Buttons*** and other ***Active Objects***, ***Data Objects Scripts***, such as ***Alarm/Event Controls***, and ***Library Object Scripts*** such as ones that might be imbedded in the ***Symbols***, or in ***ActiveX*** or ***.NET controls***.

Using Local Graphics Module Resources in an InduSoft Project

Understanding the relationship that these local scripts have on each ***Graphics Module*** is the key to understanding the Graphics Module operation… since it is only these scripts and the ***Local Scope Tags*** that allow each module to operate independently from one another; and it is the ***Server Scope Tags*** connecting them with the entire application.

Each instantiated Graphics Module (***viewer.exe***) is exactly like any other instantiated ***viewer.exe process***. This means that by project default, when the Viewer Task is set to "***Automatic***" (shown in ***Figure 2***) the anticipated first instance of ***viewer.exe*** is started when the project is placed into ***Runtime***. Additional viewers can be started on the local machine simply by using the function:

```
$WinExec($GetProductPath() & "Bin\Viewer.exe")
```

Each instantiated ***Graphics Module*** will open at coordinates "0,0" using the default project size definition (i.e., 480 x 270 to demonstrate this example) and invoke a ***Graphics Script*** unique to that particular ***viewer.exe*** process thread. In order to place a full-sized (***Start Maximized***) instantiated ***Graphics Module*** in a specific location other than the default coordinates "0,0", the following function should be placed in the ***Graphics_OnStart*** portion of the ***Graphics Script***:

```
$SetViewerPos(XStartingPosition, YStartingPosition, XProjectResolution,
  YProjectResolution)
```
[i]

If more than one viewer process is going to be instantiated on the local machine, a method for controlling where each Graphics Module is going to open must be developed within the project. In the Graphics Script ***Graphics_OnStart*** section, the following functionality must minimally be supplied:

- Keep track of each ***viewer.exe process*** that is started, by incrementing a variable as each instance is created.
- Use that Viewer ID value in a global, or server scope scheme to calculate the X and Y coordinates of where the currently starting Graphics Module (or viewer) will be located, and use those tags containing the current starting coordinates in the **`SetViewerPos()`** function.
- Save that Viewer ID value in a local tag so that it can be used to identify the local viewer process at a later time, for remotely switching screens, or remotely addressing other functionality specific to that Graphics Module.
- Using an **`Open()`** function along with the Viewer ID value in order to open a unique startup screen on each viewer, if needed or required.

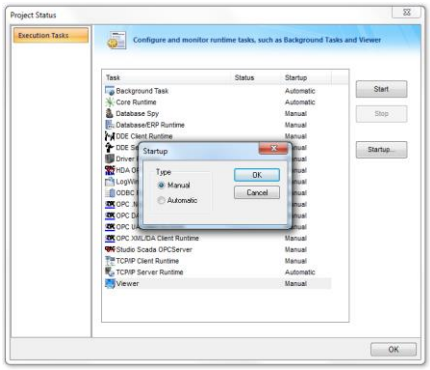

Figure 2: Viewer Task set to Manual

If more than one viewer process is being instantiated during runtime startup, ***Best Practice*** is to select "***Manual***" on the Viewer Task Startup configuration (***Figure 2***). The reason for this is due to the unpredictability of when the "***Automatic***" viewer will start, which is dependent on a number of factors including:

- How fast the local machine is
- Which script, module, or process is being serviced at the given moment
- What other tasks are being run at application startup
- Where in the list of these tasks the machine currently is when attempting to start the ***"Automatic" viewer.exe***

Tests have shown that when using the "***Automatic***" setting, it is possible that the number of viewers expected to start may occur correctly, or that one extra may also start. This is because the "***Automatic***" startup of the ***viewer.exe*** task (also starting a Graphics Script) is indeterminate when starting many viewers at the same time, which may or may not correctly increment the global process counter being used in the ***Graphics Script***. Even if the devised viewer startup scheme is linear, the "***Automatic***" startup of the default viewer is not predictable as to when it will exactly be serviced within the IWS startup background housekeeping. When dealing with the ***Startup***, ***Running***, or ***Shut Down*** of individual threads and modules, it is best to always use an assured methodology for linearly controlling events, rather than relying on a situation that is indeterminate or time dependent (e.g., adding **`Wait()`** or timers in a script) in an attempt to predict the outcome of a series of concurrent indeterminate events. Doing so will usually find a ***Use Case*** in indeterminate events where the time-dependent methodology will fail.

Tip: When using more than one ***Graphics Module*** (***viewer.exe***) to display your application on the runtime machine, ***Best Practice*** is to turn off the ***Automatic Viewer Task*** by setting the ***Viewer Task Startup*** to "***Manual***" and use the following function to open the Graphics Module(s):

```
$WinExec ($GetProductPath() & "Bin\Viewer.exe")
```

…and in the ***Graphics Script***, "***Graphics_OnStart***" Section:

```
$SetViewerPos(XStartingPosition, YStartingPosition,
    XProjectResolution, YProjectResolution)
```

Finally, there must be a way to shut down the Runtime application in a predictable manner. Running a single ***viewer.exe*** on the local machine can be handled easily by the **`ShutDown()`** function, which effectively calls an

`EndTask("Viewer")` during the Runtime Shutdown process. The issue starts to get complicated when more than one ***viewer.exe*** is running at any given time. Tests show that calling the **`ShutDown()`** function with more than one viewer running gets into the same situation as attempting to start many viewer.exe threads along with the "***Automatic***" startup of the Viewer Task. The outcome, under certain conditions due to a number of unpredictable factors, may become indeterminate and occasionally all the ***viewer.exe*** threads may not die properly; or some may lose their connection to the ***TCP/IP Server Task*** before they are killed, resulting in orphaned viewer threads, etc. In this situation, the InduSoft runtime may not "***Stop***" or may shut down only after a (long) ***TCP/IP*** timeout cascade occurs of up to several minutes in duration.

The solution to this situation is to invoke the ***DOS*** command line function **`TaskKill`** by using the function **`WinExec()`** followed by a **`ShutDown()`** function call as shown here:

```
$WinExec("Taskkill /IM viewer.exe")
$ShutDown()
```

A good way to use these functions is by placing them into a ***Global Subroutine*** and calling this from a ***Script Task*** enabled using a ***System Tag*** set from a "*Close*" Button script. This button sets this ***System Tag*** (i.e., ***bScriptExecution = 1***) as a Script Task trigger; then resetting it (i.e., ***bScriptExecution = 0***) at the end of the Script Task following the call to the ***Global Subroutine*** containing the Shut Down functions. These two functions shown above will run independently of other InduSoft Web Studio processes until they are completed. Using this scheme will insure that the function **`WinExec()`** will be invoked as exactly planned along with the function call **`ShutDown()`**. This scheme also allows the "*Close*" button to be placed in any of the independent Graphics Module screens, and it will operate with certainty.

The InduSoft Web Studio Project Environment and the Computer Desktop

Commercial Multi-Monitor "Video Wall" and "Desktop" Configurations

Multi-Monitor displays may take many forms, and involve complexity from a simple single cable connected to one external monitor, to multi-media video walls involving separate audio, video, and data for arrays of 100 or more monitors in various tiled configurations consisting of many rows with many columns. In some cases, the monitors do not even need to be of the same size, if the video wall controller is of sufficient complexity. The more complex the setup, the more likely that the monitor processor will be an independent custom-built standalone machine only requiring the input/output connections and a control panel for switching and configuration.

The minimum number of external monitors that will be directly addressed in this Tech Note is one external monitor, which is built into almost every computer motherboard today, or can be easily added on as a single video card placed into an open slot of a desktop PC motherboard. The maximum number that will be discussed is a matrix of 4 x 4 screens, as shown by example later in this Tech Note. At least one video screen manufacturer can address a 10 x 10 tiled monitor matrix with only their monitor products, requiring no external video wall controller.

Effectively, a matrix of 16 tiled identical monitors used as a video wall is simply seen from the computer's point of view as just another single external monitor, even though it has multiple screens. Such a device might find its home at a Trade Show or in a Control Room for instance, displaying digital signage, or one single screen image, or multiple independent video screens (or a combination of both), such as might be used in detention applications. With 16:9 aspect ratio monitor sizes available up to 82" (at this moment), this leaves an approximate viewing area of 4 feet high by 6 feet wide per screen. The total display size of a 16-monitor array using this monitor can approach 16 feet high by 24 feet wide maintaining the 16:9 screen ratio—large enough for most Trade Show venues. The independent controllers for tiled screen matrices allow independent inputs for each screen in the matrix, or can combine the monitors into a single screen.

The intended final use for the multi-monitor display will determine how the monitors will be configured when the matrix is assembled. In some cases, each monitor in a matrix may have a unique computer with an associated touch screen input (such as a thin client configuration) having its own independent input and output within the tiled matrix. Alternatively, the video controller could matrix all the touch panel inputs into a large single touch-screen digitizer that corresponds to the placement of the monitors beneath the individual units; this scheme providing a single touch-screen output along with a single video input. The video controller configuration will control how the monitor matrix is used—each monitor or some

number of tiled monitors up to the total number displaying a single input broken up into parts corresponding to the physical location of the monitor(s) itself (themselves).

The Advantages and Problem with Using Multiple Monitors as a Single Large Display

The main advantages in using multiple monitors as a single display are that:

1. The cost of the number of the smaller physically tiled monitors comprising the surface area of the video wall, combined with the cost of the controller are usually a fraction of the cost of a single monitor of the same physical size.
2. The video wall can be wrapped around the viewers or built in asymmetrical configurations creating a specific cinematic effect or one of picture depth.
3. The video wall can be easily broken down and transported to another location far easier than a single large screen could be.

The ***disadvantage*** is that the tiled monitors on a video wall display the mullioned single large image as disconnected segments, such as the effect of looking outdoors through a French Door, with information missing in the bezel area of the tiled monitors.

When building InduSoft Web Studio applications for tiled multiple monitor video walls, the mullion (***Figure 3***) must be taken into account, and how it is done will depend on the intent of the final application and the abilities of the video controller.

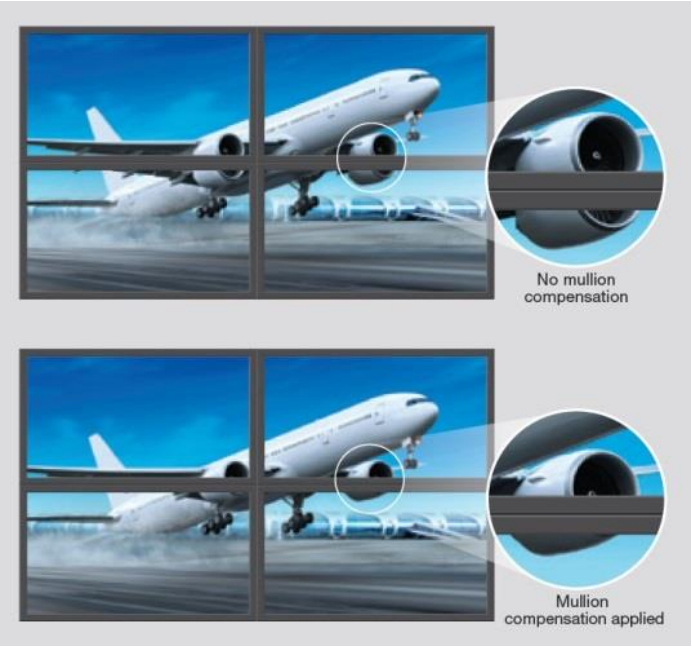

Figure 3: Mullion Compensation provided by a video controller

Addressing Mullion Issues

Virtually every type of professional external video wall controller and most, if not all, multi-monitor video cards have some scheme to address mullion issues (also called Bezel Management by some manufacturers) in tiled multiple monitor configurations. As can be seen in ***Figure 3***, a single screen image without mullion compensation will appear unnatural and chopped into pieces; while mullion compensated images actually chop off some of the image (by growing the X and Y pixels of the image, then chopping of an appropriate number equal to the mullion thickness on each edge) in each monitor the width of the bezel, which appears to "knit" all the images together into one large one. Windows Operating Systems running on a PC with one or two video ports do not attempt to address mullion issues[ii]. However, many multi-monitor

cards or external controllers that plug into the motherboard or connect via the display port, (such as the DisplayPort, Mini DisplayPort, or Thunderbolt [Apple] connectors, for instance) have some method to compensate for mullions.

Understanding Screen Resolution, Monitor Resolution, and Native Resolution

When discussing resolution, it is best to start the discussion with Screen Resolution, because this is, generally speaking, the limiting factor when determining what the final display resolution of the image desktop will be. Normally (using one external monitor as the example for this discussion), the computer resolution is aligned to the Native Resolution of the monitor, which is determined by the Monitor Manufacturer and available through the Extended Display Identification Data (EDID) provided to the computer by the monitor itself. Native Resolution is the function of the video card and driver that is directly connected to the monitor, providing image information so that image pixels line up exactly with the physical pixels of the physical monitor. This is done so that each pixel of the display image of the desktop does not display across a (calculated) percentage of a single physical pixel; thus instead deferring to the closest one, either narrower or wider than the actual image. This "smoothing" approximation effect causes a line's width (as an example) to vary according to its placement on the screen (***Figure 4***) creating fuzzy or out-of-focus-like images.

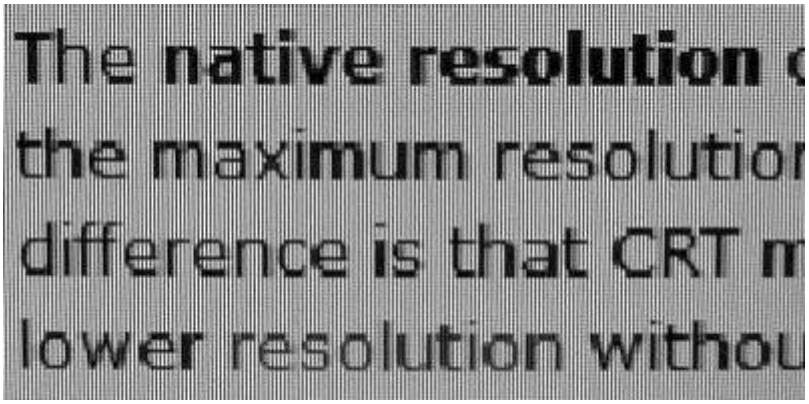

Figure 4: Close-up of an 800 x 600 image displayed on a 1024 x 786 Native Resolution monitor. The image and physical pixels do not line up all the time, so approximations are made by the monitor smoothing algorithm, creating a "fuzzy" overall appearance.

Currently, practical monitor resolutions are being tied to the aspect ratio (16:9) and resolution standards of the HD (High Definition) display, so that digital TV display standards, current DVD and Blu-Ray™ disk storage limitations, and digital computer display standards are interchangeable and usable across a wide variety of media and equipment. The current standard for HD Video 1080p translates to 1920 x 1080 in a digital computer display creating the aspect ratio of 16:9.

Most recently a new standard has been introduced called UHD (Ultra High Definition) which is intended to be displayed on monitors with extremely dense pixel populations, 4K UHD (2160p) and 8K UHD (4320p) having a monitor resolution of 7680 x 4320, with some video controller companies offering this resolution across a monitor matrix. The reasons for the upper limit at this time are twofold:

- The first reason being is that the resolution at which the human eye can discern individual pixels is being reached at normal viewing distances using the proposed 8K UHD standard, which is also approaching IMAX resolution.

- The second reason is that the amount of storage media and the transfer bandwidth needed to store/display video at these resolutions is simply impractical and costly at this moment in time. Current Ethernet wired and satellite infrastructure is not fast enough to stream 8K UHD in real-time without buffering some or the entire content first[iii].

Additionally the video controller needs a very fast processor, or parallel processors, and an impractical amount of ultra-fast video RAM in order to display the images with enough frequency to prevent image flicker or to avoid internal video synchronization issues between monitors on the wall.

Some multi-monitor video controllers can span a 4x4 (or more) matrix of 1920 x 1024 native resolution monitors, achieving an extended desktop resolution of 7680 x 4320 using a 16:9 aspect ratio on the video wall[iv]. This type of configuration encompasses the maximum native resolution of each monitor on the video wall matrix required in high-resolution venues, but the video controllers can become extremely complex and costly. The graphical environment of InduSoft Web Studio can currently support project resolutions of up to 10,000 x 10,000 pixels[v] inside of a single graphics module with no practical limitation on the placement of each unique graphics module, so the IWS software can be configured to exceed the current state-of-the-art 16:9 aspect ratio 8K UHD monitors.

"To mullion or not to mullion: that is the question" (with apologies to William Shakespeare)

When building an InduSoft Web Studio application that will be displayed on a multi-monitor array, the very first step is properly understanding the layout, and then designing the layout, allowing for any limitations. Some questions that need to be asked and answered even before the project size is defined and before any screens are built are:

1) What type of content will be displayed on the video wall? (Is the content static like signage, or live like video, or will it be a combination of media?)
2) Will each monitor or any monitors be interactive? (i.e., using touch screen input)
3) Will the video wall be configured as a single monitor, or will each monitor have a unique Graphics Module opened on it, ignoring mullions?
4) If the video wall is configured as a single continuous monitor, will each individual monitor, or will a block of tiled monitors as a subset of the video wall, display a unique InduSoft Web Studio screen?
5) What is the video wall resolution and aspect ratio?

As can be seen from the answers to these questions, that a variety of widely differing IWS applications could be designed that might accomplish similar goals, each of which becoming highly dependent on a number of factors strictly dealing with the final multi-monitor configurations. Additionally, by not properly understanding the scope of the final application and the ultimate multi-monitor display that will be used, this situation could lead to project design scope-creep and/or a poorly performing project or display.

Conclusion

When designing InduSoft Web Studio applications to display in a multiple monitor environment, it is important to understand what the final goal of the project will be, and to design that project so that the multi-monitor environment being used to display it will be fully utilized. Not understanding these factors may result in under-utilization of the monitor wall, excessive resource usage[vi] on the host computer, or other factors that were not taken into account before the application was created that could increase development time and associated application complexity.

Section 2: Designing IWS Applications to Display on Multi-Monitors

Introduction

Applications engineered to display on or across multiple monitors can involve straightforward designs that are quite simple, to complicated schemes that must keep track of many viewer modules. The following sections will discuss applications and implementations of increasing complexity, along with tips and ideas on how to create your own application that will display in the manner that you intend it to.

Case Study 1: A Simple Application/Configuration Example

Initial Configuration

The simplest and most common multi-monitor configuration is a notebook or desktop computer with an attached monitor as part of the extended desktop. The first step in configuring this environment is to establish where the external monitor is in relation to the main display. In Windows Control Panel, this means that each monitor's top edge must be at the same height in relation to each other (***Figure 5***).

Assume for the moment that this configuration has a notebook computer with a Native Resolution of 1366 x 768. This display has an aspect ratio of 16:9, or 1.78. Also, assume that you have a monitor connected to this notebook having a 1920 x 1080 pixel Native Resolution, which also has an aspect ratio of 16:9, or 1.78. Although there are more horizontal and vertical pixels in the attached monitor than on the notebook screen, both of the screens are in a 16:9 aspect ratio configuration.

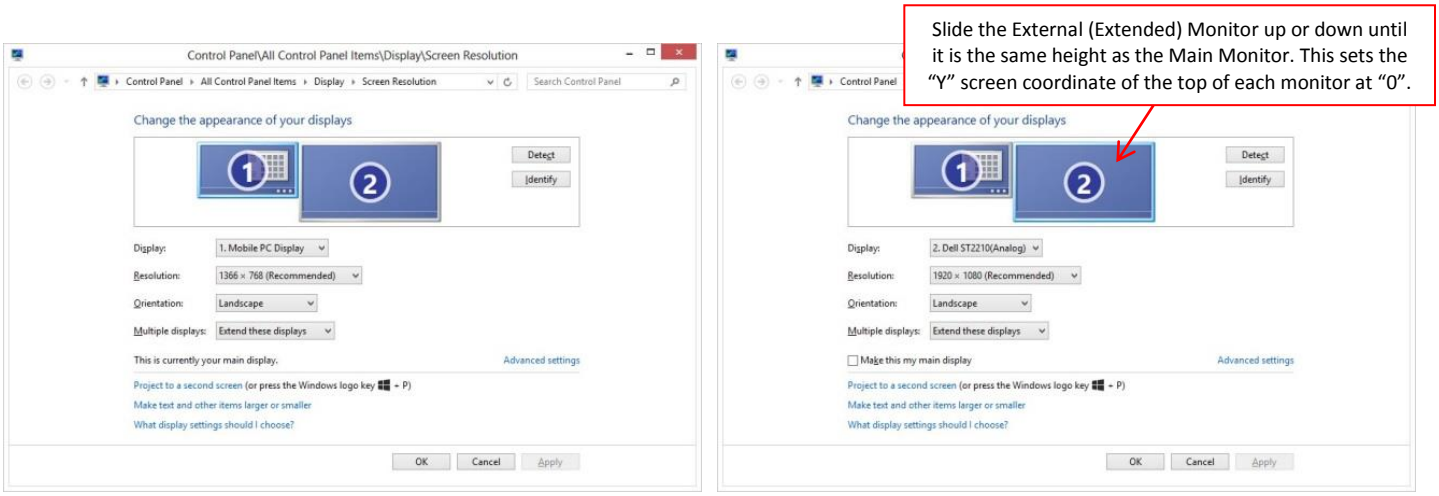

Figure 5: Multiple monitor configurations in Windows 8 OS on the same machine.

Tip: Using monitors of the same aspect ratio, but different sizes means that you can convert the project resolution (using the ***Convert Resolution*** utility) that was designed for one monitor and it will display properly at the new resolution without any object skewing (longer X-Y or Y-X relation than the original project aspect ratio).

If the application has the "***Auto Screen Scaling***" and "***Start Maximized***" attributes set in the Viewer Configurator (***Figure 6***), it will now start (assuming that the Viewer Task is set to "***Automatic***") in either monitor, and fill the entire monitor (assuming that the monitor is set as the "***main***" display in Windows). Since the aspect ratios of both monitors are the same, and by using these settings, there will not be any horizontal or vertical gray space on either monitor (***Figure 7***).[vii]

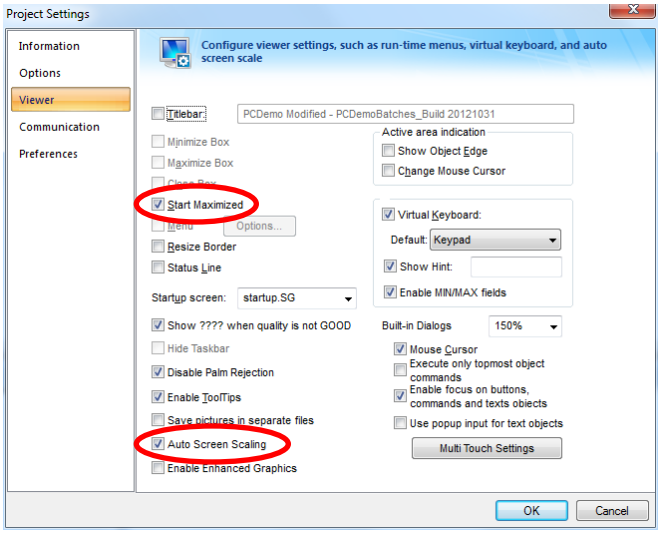

Figure 6: Viewer configurator showing the default settings "Start Maximized" and "Auto Screen Scaling" selected.

Figure 7: The InduSoft "PC Demo" application is natively built in 1280x800 resolution (5:3 or 1.6 aspect ratio). It is shown displayed on a 1366x768 monitor (16:9 or 1.67 aspect ratio) on the left, and a 1024x768 monitor (4:3 or 1.3 aspect ratio) on the right. The application is Auto-scaled to fit maximized in the monitor at either the Y-Max (left picture) leaving extra X space on the monitor, or X-Max (right picture) leaving extra Y space on the monitor.

Building "The Simple Application" and a Discussion of Viewer Module Settings

> **Note:** The following discussion starts with a simple single screen, however as the application gets more complicated, certain anticipated Runtime outcomes may not occur as expected while reading through the discussion. Several figures are shown within the discussion to help clarify these outcomes. The reader is encouraged to create or reverse engineer the included applications (which are the apps of the Case Studies below) while reading through the discussion, so that the sometimes surprising results and outcomes can be viewed firsthand and properly understood.

The Simple Application has only one screen and one viewer.exe (e.g., Graphics Module) that will be displayed on either the "***main***" or the "***external***" monitor.

A new project is opened and the size defined as 1366 x 768, which is the size of the (smaller) laptop display. This will make the Graphics Module (or the viewer.exe) the same pixel size as the computer monitor. We will create one screen called Main, the same size as the project, put a button with a ***ShutDown()*** function in it, and draw some objects in order to give the screen some perspective. This screen is set as the ***Startup Screen***, the Viewer Task Startup is set to "***Automatic***", and the "***Auto Scaling***" and "***Start Maximized***" options are checked in the Viewer Configuration.

Starting the ***Runtime*** will open the Graphics Module and the screen to the full resolution of whichever monitor is selected as the ***Main Monitor*** in Windows. Even though the project size is defined as 1366 x 768, the Graphics Module will open and scale up to 1920 x 1080, resizing everything including the fonts, when the larger external monitor is set to be the ***Main Monitor***. If the project is shown on a smaller monitor such as a 1280 x 1024 then the Graphics Module will open at the smaller 1280 x 1024 Resolution, and the screen will scale to the width of the monitor, with a grey bar at the bottom of the monitor.

By deselecting the "***Start Maximized***" checkbox, the Graphics Module will open to a somewhat smaller size than the monitor. Depending on if "***Auto Screen Scaling***" is selected or not, the screen will be resized to fit the graphics module X-maximum or Y maximum (with a grey bar to make up the difference between the screen and the Graphics Module), or open at the defined size of the screen respectively. If ***Resize Border*** is selected, the graphics module can be dynamically resized during runtime.

Notice that the ***defined font point size*** chosen for any particular object on any screen is scaled and displayed *according to the size of the Graphics Module* (***viewer.exe***) that the screen is opened on (along with all other objects on the screen), when "***Auto Scaling***" is selected. The ramification of this statement is that if screens are resized by using the ***Open()*** function, the fonts on that screen ***will not be*** resized accordingly. They will remain scaled to the size of the Graphics Module.

The final point about resizing screens during runtime is that when using the ***Open()*** function to open or resize screens, the objects on the screen will not resize (even with the "resize" option being selected in the ***Open()*** function) if that screen is currently open, *though the screen itself will be resized to the new parameters*. The screen must first be closed just prior to the Open() call with the new parameters. The objects on the screen will then be scaled to the new size (***Figure 8***).

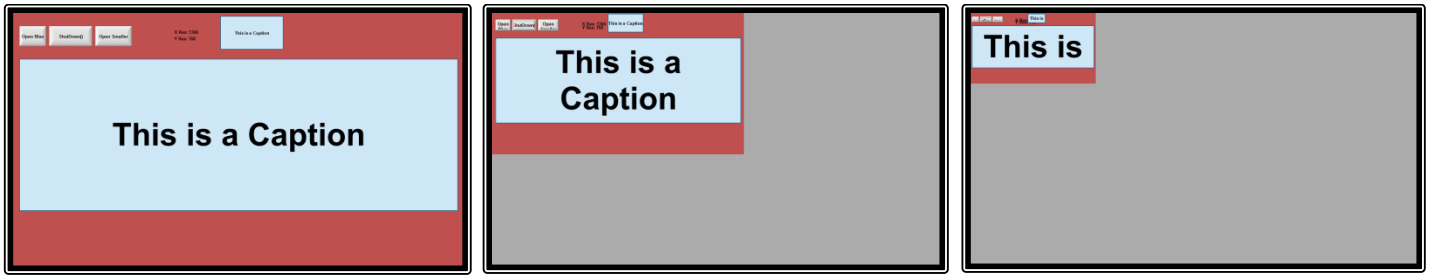

Figure 8: Three images of a resized screen. The Viewer Module is not resized (grey area), but the screen is resized using the *Open()* function. (The screen is closed before the new *Open()* function is called). The option is set to resize the objects on the screen, however, note that the text is not resized. Text size is a function of the size of the Graphics Module (*viewer.exe*)

Using these techniques, it is now possible to create a new project defining the project size as the width of the extended desktop (the width of both monitors added together) and the height as the vertical size of the bigger monitor, or in the case of monitors using vertical orientation, the height of both monitors added together and the width of the larger one.

Screens can be designed and opened which will fit within each of the monitors or a portion of each monitor by using ***Open()*** to define the location of the screen within the opened viewer.

A Slightly More-Complicated Simple Application

The next expansion step in ***The Simple Application*** is to open a Graphics Module (***viewer.exe***) on each monitor. Since the monitors are different sizes, it is necessary to specify where and what size the viewers will be. Assuming that the notebook monitor is the main display, this is how it's done (Figure 9):

1. Set the Viewer Task to start manually.
2. Create a Global Procedure that starts the viewers:

```
Sub StartViewer()
    If $ViewerNumber < 2
        $WinExec( $GetProductPath() & "Bin\viewer.exe") 'Starts viewer.exe threads
    End If
End Sub
```

3. Create a Script Task to control the call that starts the viewer(s) by setting a Boolean Tag, ***$EnableScreenStart*** in the "***Execute***" field:

```
Call StartViewer()
$EnableScreenStart = 0 'This halts the script execution by resetting the Execution Tag
```

4. In the Graphics Script, ***Graphics_OnStart()*** section, create some code to position each viewer and open a Screen to display on each viewer:

```
Sub Graphics_OnStart()
        If $ViewerNumber < 1 Then
            $SetViewerPos( 1366, 0, 1920, 1080) 'Opens the first viewer on the external monitor
            $Open("RedScreen")
            $ViewerNumber = $ViewerNumber + 1
        Else
            If $ViewerNumber = 1 Then
                $SetViewerPos(0, 0, 1366, 768) 'Opens the second viewer on the notebook monitor
                $Open("BlueScreen")
                $ViewerNumber = $ViewerNumber + 1
            End If
        End If
        $EnableScreenStart = 1 'Enables the Script Task in order to open the next viewer
End Sub

Sub Graphics_OnEnd()
        $ViewerNumber = 0 'Resets the Viewer Counter
End Sub
```

5. Add a ***Call StartViewer()*** to the ***Application Startup Script Task*** to open the first Graphics Module.
6. Finally, add a way to close the application—Place "***Call CloseViewer()***" in every Exit Button:

```
Sub CloseViewer()
    $WinExec( "Taskkill /IM viewer.exe") 'Kills all viewer.exe threads
    $ShutDown() 'Shuts the application down normally
End Sub
```

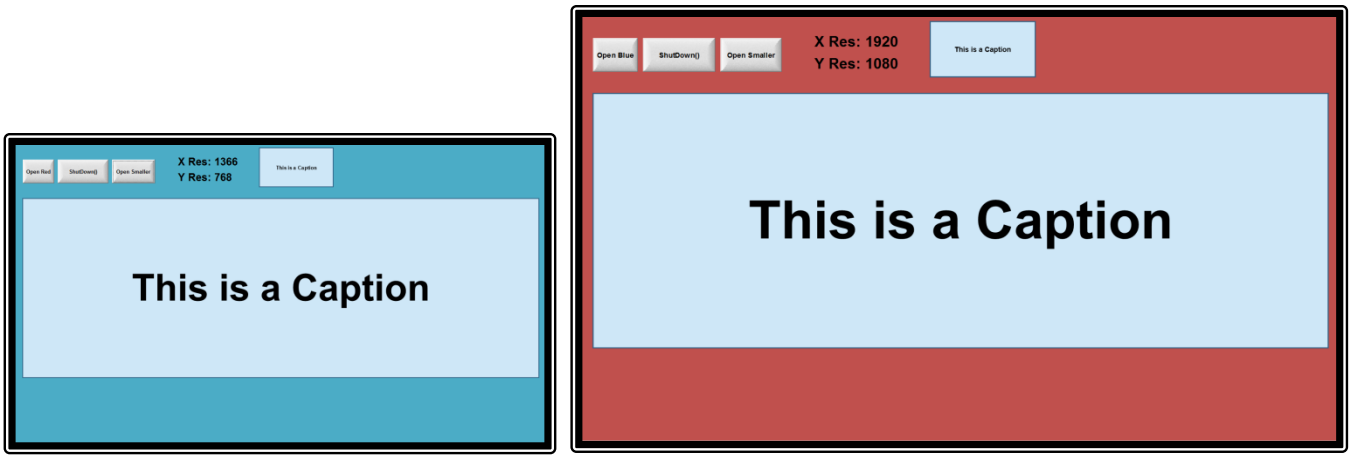

Figure 9: Identically sized project screens (1366 x 768) using identically sized objects and text point sizes, opened full-screen on two monitors that are different sizes within the same project and Runtime.

Case Study 2: Creating a Multi-Monitor Array Application using One Graphics Module

Now that the fundamentals of how the Graphics Module behaves and reacts have been demonstrated, a new application can be created that will display across a multi-monitor array. This application will use a single Graphics Module filling out the resolution of the monitor array. This configuration and application are suitable when small individual screens displaying on each monitor or across several monitors in the array need to be used and there is no requirement to provide individual touch inputs for each monitor. Additionally, it is easy to create single screens that will span across part of, or across the entire monitor array. Mullion compensation should be a factor of the video controller, since creating, sizing, and opening screens that take into account mullion compensation may be tricky and complicated. The issues of needing to adjust individual screen sizes and placement may be time consuming and will depend on the content being displayed and the complexity of the monitor array.

Obviously full-screen video, images, or moving content across the entire array will display best if mullion compensation is used, especially since any of this media will likely be shown on one single large screen the size of the monitor array resolution or possibly on a subset of the entire array[viii]. However, the content that is not displayed because it is "behind" the mullions may be important, or it may be incidental, and deciding whether or not it is one or the other is a matter of content design and display needs. In any case, your application should be viewed on the final monitor array and compensated for well before "Showtime" occurs.

Building the Application

Assume that the video controller for this example has a resolution of 1920 x 1080 and that a tiled monitor array matrix of 4 x 4 (16) monitors are being used.

Additionally for this discussion, the project definition will be 1920 x 1080. If needed, a smaller project size could be chosen especially if it helps facilitating the creation on the development machine (e.g., the final display resolution being on the order of 8K UHD [7680 x 4320]), as long as the aspect ratio is the same; 16:9 or 1.67. Developing full screens in 8K UHD resolution will be troublesome on a quarter-sized or less monitor where the entire layout may not be able to be fully or appropriately rendered during development.

As shown in the previous sections, a Graphics Module can be scaled up to a larger display resolution if necessary; however, trial-and-error checking during development will help arrive at a suitable screen size and project/screen/monitor resolution or size. Font point sizes will be selected as appropriate for the particular screen(s) that they are being displayed on. In order for information on smaller screens to be seen from a distance, proportionally larger font sizes must be used on them.

There are two main types of screens that need to be created for this example application, with one optional screen size:

1) Single full sized screens the size of the monitor array, or 1920 x 1080.
2) Small Screens, ¼ the size of the full display. These screens will be 480 x 270 which is ¼ of the full display resolution of 1920 x 1080.
3) ***Optional:*** Screens sized as a subset of the larger array encompassing, for instance, four adjacent monitors in a 2 x 2 tiled configuration. These are essentially built and opened the same way that screens in (2) are, except that they are twice as big.

The following discussion is considerably more complicated than the ***Simple Application***. The reason is that the screens, whether they are individual screens or whether they are instances of the same screen *will need to be directed to open in specific locations, and possibly in specific sizes* (e.g., when using mullion compensation and displaying a slightly shrunken complete screen within the viewing area of a single monitor). In the case of opening instances of the same screen, each instantiation will additionally need to be given *a unique awareness of itself through its screen name, and what the instance ID number is*. The uniqueness characteristics are passed to the particular screen using custom properties, called Mnemonics[ix]

The Single 1920 x 1080 Screen

The application is started by creating a single large screen which will span the entire monitor array. This demo application shows a simulated full screen .NET Web Browser or Media Player by using an image the size of the screen as a background image. Copies of the screen can be made with simulated outlines of the monitor locations and mullions as needed for registration, alignment, and placement purposes. These main screens also serve to host the buttons which open all the screens of the demo application (***Figure 10***).

Figure 10: These three 1920 x 1080 screens host the buttons and are identical except for the simulated monitor locations and mullions.

The 480 x 270 Screens

Next, a 480 x 270 Demo Screen needs to be built, which will have registration information on it (***Figure 11***). Sixteen of these screens can be tiled into the space of a 1920 x 1080 monitor, which corresponds to each monitor position tiled in the multi-monitor array. The useful thing about building a screen like this for your application is that you can view the final result on the multi-monitor array as an alignment and calibration tool (***Figure 12***).

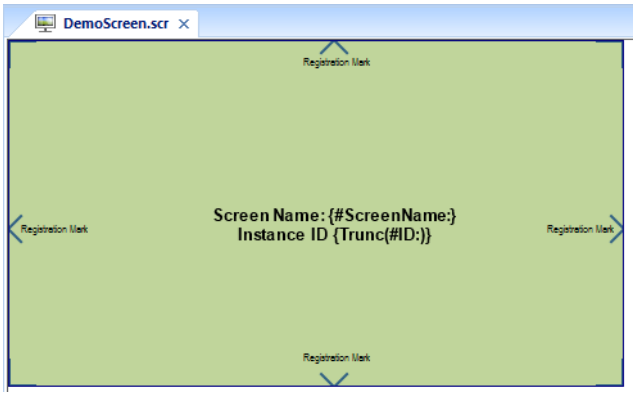

Figure 11: Demo Screen showing the 1-point border, 2-point registration marks, and labels displaying self-awareness using custom properties and mnemonics.

The screen is constructed using a contrasting background color, a border rectangle made with a 1-point line, centered 90°2-point registration marks, and 90° 2-point corner bracket registration marks which will outline the screen and show placement. Use of the "***Format***" Tab and the "***Size***" and "***Position***" sections will facilitate creating these items accurately. Defining this screen as a popup insures that any screens it opens on top of will not close. No screen border is configured in the ***Screen Attributes*** since the 1-point border was already drawn as part of the registration marks.

Many schemes could be devised for defining the screen locations on the fly, however for this example, a Class Tag is defined and initialized in the Startup Script Task, which holds the X and Y positions of each screen position, along with the position ID number as shown here, which makes defining the location of each screen easy to understand and follow. Any screen location can be accessed simply by using the correct ***$cScreen[]*** index. This will be useful when building more than one 480 x 270 screen and opening it in the correct location simply by knowing the target monitor position (ID number):

```
Dim count: For count = 0 To 15      'Initializes the 16 Screen-Class Array Tag members - iXPos, iYPos, and iScreenNum
                $cScreen[Count].iScreenNum = Count
                $cScreen[Count].iXPos = $Mod(Count,4)*480
                $cScreen[Count].iYPos = $Trunc(Count/4)*270
        Next
```

The Demo Screen can be opened in all sixteen monitor locations by using an On-Down VB Script in a Button which calls an ***OpenScreens()*** Subroutine:

```
$sScreenName = "DemoScreen"      'Initializes the $ScreenName Tag with the name of the screen that will be opened
Call OpenScreens()
```

The Subroutine ***OpenScreens()*** is located in the ***Global Procedures*** and opens ***DemoScreen*** instances in sixteen unique locations (***Figure 12***) using the values stored in the ***$cScreen[]*** Class Tag Array.

Additionally, in the ***$Open()*** statement, the use of mnemonics is used to assign the custom properties, ***#ScreenName:*** and ***#ID:*** to each opened screen, giving each screen instantiation uniqueness that can be accessed for the ***$Close()*** function to work. Since ***$iCounter*** is a numeric tag and not a string value, special syntax is used to assign it to the #ID: custom property[X]. Finally, unique mnemonic names must be used for each screen that is being opened (e.g., #IDScreen1:, #IDScreen2:, etc.). You cannot use identical mnemonics with two different screens.

```
Sub OpenScreens()   'This routine will open 16 identical screens according to the positions stored in the $cScreen[] Class Tag
        Array
    Dim i:For i = 0 To 15
                $Close($sScreenNameOld, i)
                $iCounter = $cScreen[i].iScreenNum
                $Open($sScreenName, $cScreen[i].iXPos ,$cScreen[i].iYPos ,0,0,0,$cScreen[i].iScreenNum,
                        "#ScreenName:sScreenName #ID:" & $iCounter)
        Next
End Sub
```

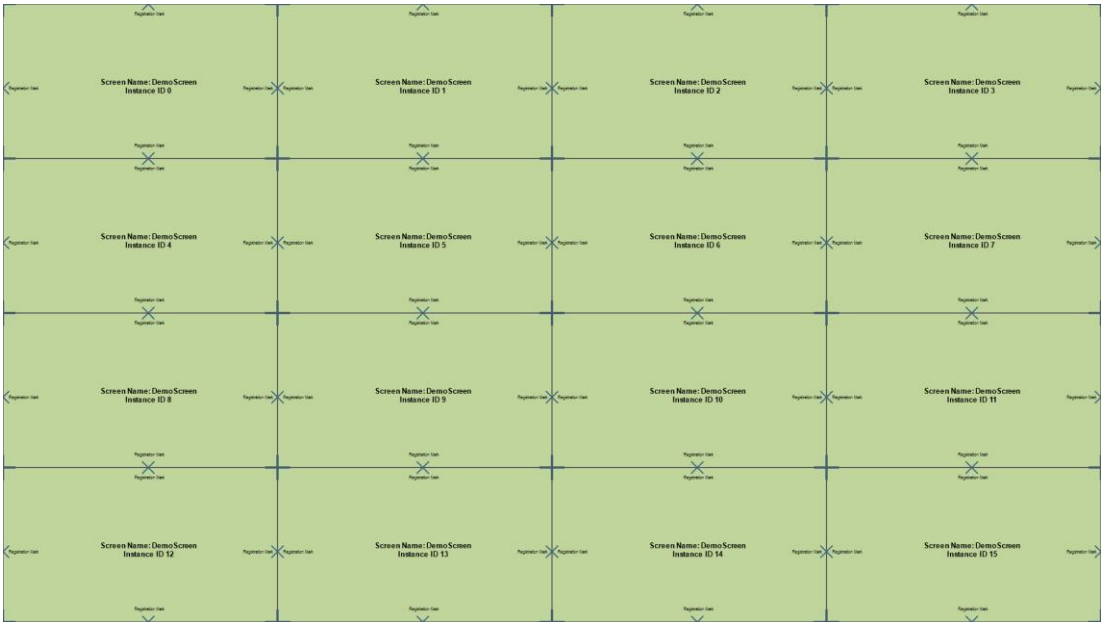

Figure 12: Sixteen *DemoScreen* instances opened on a 1920 x 1080 monitor (array). Note the unique ID number of each instance.

Finally, the last thing to do with the Demo Screen is to place a transparent rectangle covering most of the surface of the screen with a Command Link. Use an On-Down VB Script with a ***$Close()*** function addressing the mnemonics, which closes the screen (itself) that is being clicked on. Move the rectangle to the back before saving the screen:

```
$Close($#ScreenName:, $Trunc($#ID:))   'This closes the screen instances that know their own identity.
```

This screen can be copied and the border rectangle changed to show mullion simulation, which can help to give a perspective on how the final application will display on the tiled monitor array during development on a smaller monitor. The mullions are created by placing a rectangle with no fill using a dark grey 10-point line size, and on top of it, another rectangle with no fill, using a light grey 2-point line size (***Figure 13***).

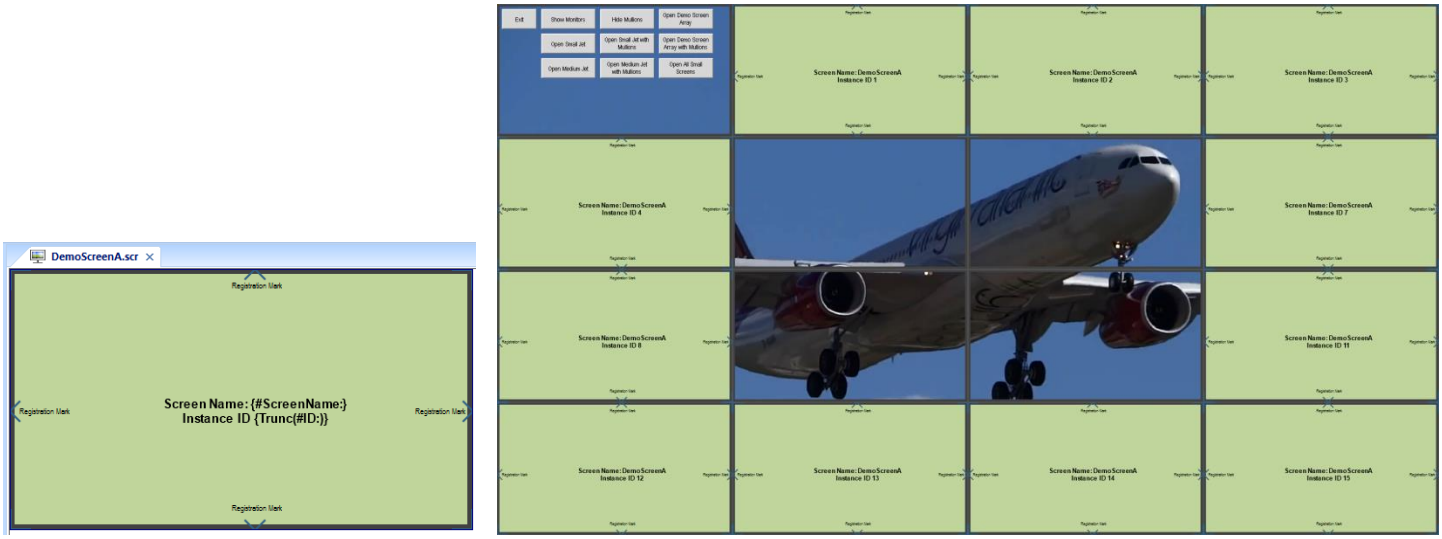

Figure 13: Single Demo Screen showing mullions and registration marks. Pushbuttons located on the main screen with mullions added (underneath the Demo Screen popups) open the screens or screen groups. Screens are closed by clicking on them.

Case Study 3: Creating a Multi-Monitor Array Application using Many Graphics Modules

The final demo application that will be discussed involves using a notebook computer and external monitor, as previously discussed. Sixteen separate Graphics Modules (***viewer.exe***) are instantiating in the external 1920 x 1080 monitor with a seventeenth Graphics Module used as a remote control panel for the others (***Figure 14***) located on the notebook main display.

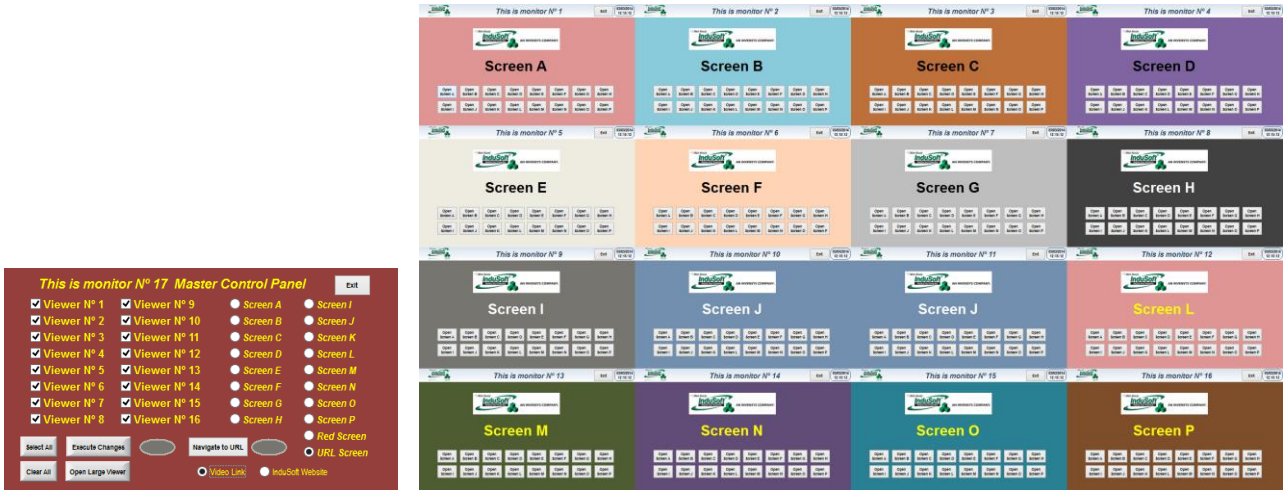

Figure 14: Multiple Viewer Demo Application after startup. Each Viewer's screen in the array is made up of a header screen and a selection screen where any of the other screens in the app can be selected.

The advantages of using this technique are that each viewer is independent of the others and each one has its own set of local graphics scripts and tags. This configuration therefore allows each viewer to be displayed on its own platform if required, whether physical or virtual as a Remote Thin Client with each screen in this configuration having its own input, such as a touch screen input or digitizer.

This configuration can be used in interactive displays where a wall of touchscreen monitors with each screen and input communicating through the local machine to the runtime server. Such a configuration could find use at a tradeshow

showing interactive signage or a map of various facilities displayed on the video wall. Touching a specific facility would zoom that facility to full scale or perhaps display a menu of other options within the chosen monitor. Many other uses for this configuration could be found in detention and control room facilities.

Additionally, this configuration can be used where mullion compensation is not used or needed since the screens are opened and resized according to the size/position of the opened Graphics Module. In this configuration, text sizing is not an issue. It is possible to open/close Graphics modules if a scheme to retrieve and store the ***PID*** of each particular viewer.exe were developed. Initial testing has shown that opening a new or resized Graphics Module and retrieving the ***PID*** of the last ***viewer.exe*** module using the command line ***TASKLIST***, then using that ***PID*** in a ***KILLTASK /PID xxxxx*** command will shut down the last opened monitor. It seems prudent however to develop a more precise way to know which viewer module is being addressed with the KILLTASK command, since the test for this discussion was done only as a proof-of-concept (***Figure 15***).

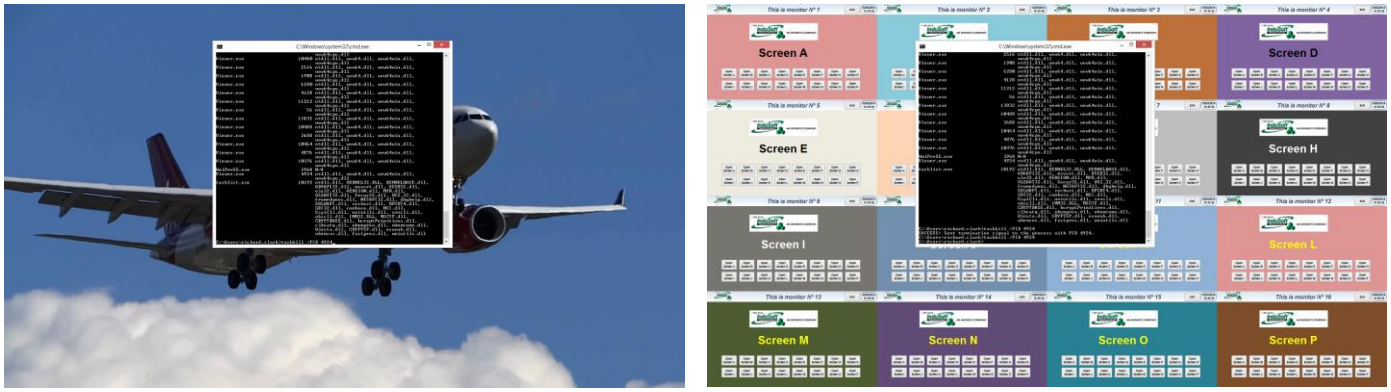

Figure 15: Opening a large viewer across the monitor array and then killing it using TASKKILL and the PID of the process thread.

Screen groups are created using the Header.scr and a unique screen name from ScreenA.scr through ScreenP.scr. These screen group names are saved to a string array for access later during the application startup. The value on the header screen is a local string tag that is created on the fly as each viewer starts up, containing the string “This is Monitor Nº ” concatenated with the Viewer Number. As viewers are instantiated, the local graphics script also assigns a local tag the $ViewerID number, therefore each viewer has a tag that knows what its ID number is. The opening screen group for each viewer is equal to the index of the string tag array holding the screen group name.

The selectors on the Master Control Panel (Monitor 17) work by selecting one or more monitors which are encoded as bits into an integer tag, $ViewerSelection. The selected screen is saved in the tag $Screen Selection as the screen number. When the “Execute Changes” button is pressed (i.e., on down), the server tag $ExecuteChanges is set, and when released (i.e., On Up) the same tag is reset. Setting the tag causes a local execution tag (LocalChangeExecution) to be set in the local graphics script while running section. When the button is released, the values do not match anymore in the local graphics script, which causes a one-shot execution to test to see if the current viewer is being addressed (using the $GetBit() function), and if so, opens the screen number stored in $ScreenSelection.

The buttons on the Master Control Panel do the following:

- **Select All**: Selects all Viewers (puts 65535 into $ViewerSelection)
- **Clear All**: Clears all Viewers Selected (puts 0 into $ViewerSelection)
- **Execute Changes**: Operation and sequence is discussed in the previous paragraph.
- **Open Large Viewer**: Starts Viewer 18 covering all viewers in the external monitor with a single screen. The only ways to close this screen are to either exit the runtime (push Exit on the Master Control Panel) or use a command line TASKKILL as described earlier in this section (Figure 15).

- **Navigate to URL**: Sets the method of the .NET Web Browser Control on the URL Screen(s) to either of the selected URLs under the button. Press the button for about 1 second. There may be a delay of a few seconds during the navigating to the Website or the video. If all screens do not navigate, press and hold the button again for one or two seconds.

- **Exit**: Kills all viewer threads and exits the InduSoft Web Studio Runtime.

Here are the scripts:

```
'Startup Script:
        $StartupScreen[0] = "ScreenGroupA.sg"
        $StartupScreen[1] = "ScreenGroupB.sg"
        $StartupScreen[2] = "ScreenGroupC.sg"
        $StartupScreen[3] = "ScreenGroupD.sg"
        $StartupScreen[4] = "ScreenGroupE.sg"
        $StartupScreen[5] = "ScreenGroupF.sg"
        $StartupScreen[6] = "ScreenGroupG.sg"
        $StartupScreen[7] = "ScreenGroupH.sg"
        $StartupScreen[8] = "ScreenGroupI.sg"
        $StartupScreen[9] = "ScreenGroupJ.sg"
        $StartupScreen[10] = "ScreenGroupK.sg"
        $StartupScreen[11] = "ScreenGroupL.sg"
        $StartupScreen[12] = "ScreenGroupM.sg"
        $StartupScreen[13] = "ScreenGroupN.sg"
        $StartupScreen[14] = "ScreenGroupO.sg"
        $StartupScreen[15] = "ScreenGroupP.sg"
        $StartupScreen[16] = "Red.scr"
        $StartupScreen[17] = "URL.scr"
        $ViewerNumber = 0
        $ViewerSelection = 0
        $ViewerSelectionLocal = 0
        $ScreenSelection = 0
        $DisposeMethod = 0
        Call StartViewer


'StartScreen() Script Task
        Execution $EnableScreenStart = 1
        'The code configured here is executed while the condition configured in the Execution field is TRUE.
        Call StartViewer() 'Calls the Global Procedure to Start the Viewer Opening process
        $EnableScreenStart = 0


Sub StartViewer() 'Global Procedure to start each of the viewers
        If $ViewerNumber < 17 Then
                $WinExec($GetProductPath() & "Bin\Viewer.exe")
        End If
End Sub


'Button Script: Open Large Viewer
 $WinExec($GetProductPath() & "Bin\Viewer.exe")


'CloseScreen() Script Task
        Execution $EnableScreenClose = 1
        'The code configured here is executed while the condition configured in the Execution field is TRUE.
        Call CloseViewer() 'Calls the Global Procedure to which closes the app
        $EnableScreenClose = 0


Sub CloseViewer() 'Global Procedure to Kill all viewer threads and shut down the app
```

```
        $WinExec( "Taskkill /IM viewer.exe" )
        $Shutdown()
End Sub

'Graphics Script
'This procedure is executed just once when the graphic module is started.
Sub Graphics_OnStart() 'This opens the first 16 viewers into positions calculated on the fly
        $Sys.ResolutionX = $GetAppHorizontalResolution()
        $Sys.ResolutionY = $GetAppVerticalResolution()
        If $ViewerNumber < 16 Then
                $SetViewerPos(1366 + ($Mod($ViewerNumber,4))*($Sys.ResolutionX),
                        (($Trunc($ViewerNumber/4))*($Sys.ResolutionY)), $Sys.ResolutionX, $Sys.ResolutionY)
                $Open($StartupScreen[$ViewerNumber]) 'Opens the screen group that is associated with the viewer
                        number as initialized in the Startup Script
                $Title = "This is monitor N" & ($ViewerNumber + 1)
                $ViewerNumber = $ViewerNumber + 1
        Else
                If $ViewerNumber = 16 Then 'This opens the Master Control Panel
                        $SetViewerPos(400, 100, 480, 270)
                        $Open("MasterControl")
                        $Title = "This is monitor N" & ($ViewerNumber + 1) & "  Master Control Panel"
                        $ViewerNumber = $ViewerNumber + 1
                Else
                        If $ViewerNumber = 17 Then 'This opens the Large Graphics Module with the Jet. The viewer.exe
                                is started by pushing the button
                                $SetViewerPos(1366, 0, 1920, 1080)
                                $Open("Full Screen1")
                                $Title = "This is monitor N" & ($ViewerNumber + 1) & "  Master Control Panel"
        '                       $ViewerNumber = $ViewerNumber + 1
                        End If
                End If
        End If
        $ViewerID = $ViewerNumber
        $EnableScreenStart = 1
End Sub


'This procedure is executed continuously while the graphic module is running.
Sub Graphics_WhileRunning()

        If $ExecuteChanges = 1 Then 'This sets the local tag to start the screen changing execution
                LocalChangeExecution = $ExecuteChanges
        End If


        If LocalChangeExecution <> $ExecuteChanges Then 'When the button is released, the execution tags don't match
                anymore and this routine is run
                If $GetBit( $ViewerSelection, $ViewerID-1 ) = 1 Then 'This gets the screen number from the selection and
                        see if it is itself, if so, then this runs
                        $Open($StartupScreen[$ScreenSelection-1])  'Opens the newly selected screen
                End If
                LocalChangeExecution = $ExecuteChanges 'Resets the local execution tag
        End If
End Sub
```

Appendix

External References and Supplemental Reading

InduSoft, Inc. ***IWS v7.x.x.x Technical Reference Manual (.chm help)***. Latest Release is available at: http://www.indusoft.com/Documentation

InduSoft, Inc. ***IWS v7.1+SP2 Technical Reference (.pdf help)***. Available in US Letter, A4 formats, and Chinese is available at: http://www.indusoft.com/Documentation

InduSoft, Inc. ***Driver Runtime Tech Note***. Available at: http://www.indusoft.com/Documentation/Technical-Notes?EntryId=141&Command=Core_Download

InduSoft, Inc. ***MultiMonitorProjects.zip***. Contains three sample applications discussed in this Tech Note as Case Studies. Download at:

http://www.indusoft.com/Products-Downloads/Sample-Applications?EntryId=1070&Command=Core_Download

Also available for download on the InduSoft Web Page Sample Applications Downloads Section.

Tech Note Revision Table

Revision	Author	Date	Comments
A	Richard Clark	March 3, 2014	Initial version Ok'd for publication by M. Gadbois
B	Richard Clark	March 27, 2014	Spelling Corrections; added link for Sample Applications/Case Studies

Endnotes

[i] See Indusoft Web Studio ***Help Manual*** for more details about these functions.

[ii] As of this publishing date; including current Windows 7 and Windows 8 versions with the current set of default video drivers that are provided with the OS.

[iii] See: http://en.wikipedia.org/wiki/8K_resolution

[iv] As of this publishing date.

[v] Assuming that the desktop can support the resolution. It is not possible (at the time of publication of this document, in InduSoft Web Studio versions 7.1.2.3 and prior) to open viewers at location coordinates less than zero. ***This feature is anticipated be added in a future Service Pack or patch*** (check your currently installed version for this functionality which will supersede this ***footnote***).

> **Issue Description**: If you have a left and right monitor on each side of your Main Display, along with an upper row of monitors (example: a 3 x 2 tiled configuration consisting of six monitors with the Main Display at the lower-center location, all configured as an extended desktop); the upper left corner of the Main Display is at coordinates 0,0. InduSoft Web Studio Graphics Modules cannot be opened on the left monitor or in the upper row of monitors using negative coordinates. Until this functionality is available, the Main Display must be in the Upper Left position of the monitor array in order to have access to the entire extended desktop of all the displays.

[vi] See the blog http://www.indusoft.com/blog/2014/01/24/using-the-microsoft-net-webbrowser-2-0-control-in-indusoft-web-studio/ for resource usage considerations using the .NET Web Browser Control in multiple instantiations of the Graphics Module on the same machine.

[vii] More information on using aspect ratios in InduSoft Web Studio projects can be found in this blog posting: http://www.indusoft.com/blog/2013/10/01/aspect-ratios-and-your-indusoft-web-studio-project/

[viii] This type of configuration could be useful in a control room video wall application, such that there were multiple cameras, each assigned to a monitor being displayed on an embedded ActiveX or .NET video viewer. By sizing the OCX object to fit within the viewing area of the monitor (even though the screen edges are hidden under the mullions) is quite easy, if the screen it is embedded in is simply defined as the default size of the monitor footprint. A ***Command Link*** can be added to this display which will open a larger screen with the same OCX on it, opening across several or all of the monitors on the wall, showing a larger view and close up of the selected camera. Additionally, if there are full sized screens (built for no mullion compensation, such as an HMI display) that need to be shown within a single monitor of a mullion compensated video wall, the screen could be opened and resized a little smaller (text will not be resized), that will completely show within the viewing area of the monitor, if showing hidden information along the edges is important. It may be possible with some sophisticated video controllers to use “dynamic mullion control” that will shut off mullion compensation on a single monitor, synchronizing this feature with the screen being displayed, so that screen distortion using the “open()” function is not an issue for these screens which otherwise would need to be shrunk when the mullion compensation is on.

[ix] See the ***Open()*** function discussion in the Indusoft Web Studio ***Help Manual*** (e.g., ***Technical Reference***) for a detailed explanation of the Mnemonic option.

[x] The expected syntax to assign the ***#ID:*** custom property is ***“#ID:$iCounter”*** in the mnemonic assignment area of the ***$Open()*** function. However, in the current IWS version 7.1.2.3, assigning numerical values to a mnemonic requires special syntax as shown in the example. This may be changed in a future version of InduSoft.