Getting Started

Before writing any applications that use *Krypton* you are recommended to read the documentation in order to speed up your development and get the most from the suite of components. To explore some of the capabilities you should run the *Krypton Explorer* application that was placed onto the desktop during installation. You can then explore the various example applications.
As a minimum you should read the Overview - Structure section to get a feel for the common organization of all the controls and components. Then run through the Tutorials - Three Pane Application tutorial for an example application being built with the controls. Finally read the description of a typical control such as Toolkit - KryptonButton to see how specific controls are structured.

- Overview
  - Structure
  - Background
  - Border
  - Content
  - ButtonSpec
  - Palettes

- Toolkit
  - KryptonBorderEdge
  - KryptonBreadCrumb
  - KryptonButton
  - KryptonCheckBox
  - KryptonCheckButton
  - KryptonCheckedListBox
  - KryptonCheckSet
  - KryptonColorButton
  - KryptonComboBox
  - KryptonCommand
  - KryptonContextMenu
  - KryptonDataGridView
  - KryptonDateTimePicker
  - KryptonDomainUpDown
  - KryptonDropButton
  - KryptonForm
  - KryptonGroup
  - KryptonGroupBox
  - KryptonHeader
  - KryptonHeaderGroup
  - KryptonInputBox
  - KryptonLabel
  - KryptonLinkLabel
  - KryptonListBox
  - KryptonManager
  - KryptonMaskedTextBox
  - KryptonMessageBox
  - KryptonMonthCalendar
  - KryptonNumericUpDown
  - KryptonPanel
  - KryptonPalette
  - KryptonRadioButton
  - KryptonRichTextBox
  - KryptonSeparator
  - KryptonSplitContainer
  - KryptonTaskDialog
  - KryptonTextBox
  - KryptonTrackBar
  - KryptonTreeView
  - KryptonWrapLabel
  - Custom Controls
    - Using IPalette
    - Using IRenderer

- Navigator
  - Overview
  - KryptonPage
  - Modes
  - Page Dragging
  - PopupPages
  - Tooltips
  - Other Properties
  - Bar Modes

- Ribbon
  - Overview
  - Tabs
  - Groups
  - Group Containers
  - Group Items
  - Contextual Tabs
  - Application Button
  - Quick Access Toolbar
  - KeyTips & Keyboard Access
  - ButtonSpecs
  - Control Events
  - Item Events
  - KryptonGallery

- Workspace
  - Overview
  - Layout
  - Compacting

- Button Modes
- Group Modes
- Header Modes
- Outlook Modes
- Panel Modes
- Stack Modes
- Selection Events
- Action Events
- Other Events

Docking
- Getting Started
- Page Creation
- Flags
- Persistence
- Hierarchy
- Persistence Events
- User Requests Events
- Controls Events
- Drag & Drop Events

Page Dragging
- Overview
- DragManager
- Drag Enabling Controls

Tutorials - Toolkit
- Using Krypton in VS2005
- Using Images with Buttons
- Embedding Palette Definitions
- Multiple Choice Buttons
- Three Pane Application
- Expanding HeaderGroups (Splitters)
- Expanding HeaderGroups (DockStyle)
- Expanding HeaderGroups (Stack)

- Sizing
- Workspace Persistence
- Page Dragging
- Events

Tutorials - Navigator
- User Page Creation

Overview

Before using the *Krypton* suite of components you should read all of the overview section.

Structure
Describes the consistent organization used for each of the controls and components. Once you understand how they are structured it becomes much easier to quickly and easily find the property or event you need to modify. At a minimum you should read this section before using the controls in your own application.

Background
Describes the properties used to customize the drawing of the background. Not all controls draw a background and so this set of properties will only be found within the states of those controls that have a background drawing capability.

Border
Describes the properties used to customize the drawing of the border. Most but not all of the controls have the ability to draw a border and so the set of properties are only found in the states of controls that draw a border.

Content
Describes the properties used to customize the drawing of control values. Any control that draws values such as text will have a set of content properties within each control state.

ButtonSpec
Describes how button specifications can be used to add button functionality to some of the *Krypton* controls.

Palettes
Describes how to design and use palettes within your application in order to alter the look and feel quickly.

Control Structure

All of the *Krypton* controls follow the same basic structure in the organization of properties and how features are exposed. Once you understand the standard organization you can quickly find and customize the control feature of interest. Figure 1 shows the set of properties exposed by the *KryptonButton* control that will be used as the example for the rest of this section.

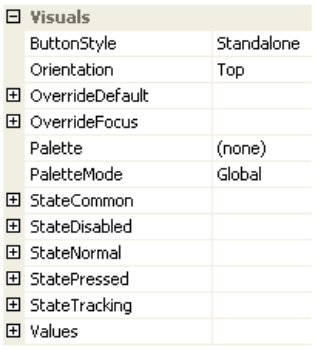


Figure 1 – KryptonButton standard properties

Notice that the entire standard set of properties are grouped together under the category name of *Visuals*. If you ensure your properties window in *Visual Studio* is organized by category then it becomes very easy to find this group.

Not all *Krypton* properties are exposed by this category, only the standard set. For example the *KryptonSplitContainer* has a *SplitterWidth* property that is placed in the *Layout* category. This maintains consistency with the property of the same name that is also in the *Layout* category for the *Windows Forms SplitContainer* control.

Styles
Each type of control has multiple styles that are used to determine how the control will be displayed. This style value is exposed as an enumeration property and in the case of our *KryptonButton* example it is called *ButtonStyle*. The different styles can be thought of as the different variations available for that control.

In the case of the *KryptonButton* the *Standalone* style is used for a standard button that is appropriate for most situations.
The *LowProfile* style is instead intended for use in scenarios where the button should blend into the background of the container until the user interacts with it by giving it focus for moving the mouse over it. Figure 2 shows an example of the button using each of these two styles.

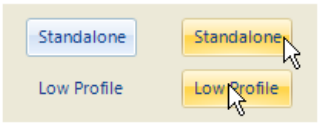


Figure 2 – ButtonStyle examples

The first column shows the *Standalone* and *LowProfile* styles when the button is passive. You can see that the second button style has a transparent background in order to be less intrusive. In the second column you can see that when the user moves the mouse over the buttons they both then become highlighted in the same way.
A third button style of *ButtonSpec* is used for buttons when they are placed inside of other *Krypton* controls and the fourth style of *Form* is for caption buttons that are used by the *KryptonForm*. There are also extra custom styles that can be defined by using the KryptonPalette component. These extra styles always start with the word 'Custom' and end with a number. In the case of button styles there are three with the names *Custom1*, *Custom2* and *Custom3*. Styles for different controls have a differing number of custom entries available.

Palette
When any of the *Krypton* controls is drawn the details are sourced from a palette that contains all the default values for all of the different controls. This allows you to quickly switch the appearance of the controls by switching to using a different palette. Note that the palette is used to recover the values to use when they have not been customized at the per control level. See details of the *State* and *Override* properties for more details about customizing an individual control.

Each control has two properties called *Palette* and *PaletteMode* that define which palette to use. *PaletteMode* determines how to find the palette. The default value is called *Global* and means that the control will use the single global palette. Most applications want all of the *Krypton* controls to have a consistent look and feel. In order to support this scenario there is a global palette setting. If you change the global palette then all the controls with a *PaletteMode* of *Global* will automatically start using this new value.

To change the global palette you need to add an instance of the *KryptonManager* component to your application and then modify the *GlobalPaletteMode* and *GlobalPalette* properties. You can do this at design time by just dragging the *KryptonManager* component from the *Toolbox* and dropping it on your *Form*. Then select the new component and search in the properties window for the

GlobalPaletteMode and *GlobalPalette* properties.

Instead of using the *Global* value you might prefer to specify one of the built-in palettes for use with the control instance. In this case you just need to select one of the entries such as *Professional - System* and the control will then be fixed to using that built-in palette.

You are not limited to using just built-in palettes. If you use a *KryptonPalette* component in your application then you can create your own custom palette with whatever settings you like. Just drag and drop the *KryptonPalette* from the *Toolbox* onto your *Form* and then use the properties window to setup the custom values. In order to modify your control to use the custom palette you need to use the *Palette* property of the *Krypton* control. Once assigned, which can be done at design time using the properties window, the *PaletteMode* value will automatically change to *Custom*. Figure 3 shows three *KryptonButton* instances where the first two are using built-in palettes of *Professional - Office 2003* and *Professional - System* and the third is using a *KryptonPalette* that has been customized.

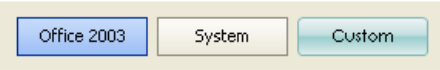


Figure 3 – KryptonButton with different palettes

Renderer
All control drawing is performed by a renderer. Unlike the palette you cannot alter the renderer used on a per control basis but instead can only alter it at the global level. To change the global renderer you need to add an instance of the *KryptonManager* component to your application and then modify the *GlobalRenderMode* and *GlobalRenderer* properties. You can do this at design time by just dragging the *KryptonManager* component from the *Toolbox* and dropping it on your *Form*. Then select the new component and use the properties window to change the settings.

State
Each control has a number of properties that begin with the word *State*, such as *StateDisabled* and *StateNormal*. These are the properties you use to customize the appearance of the control based on the state of the control.

Each control has the two states *Disabled* and *Normal*. Some controls have additional states, for example the *KryptonButton* also has *Tracking* and *Pressed* states which have matching customization properties of *StateTracking* and *StatePressed*. To work out when each state is applied you should refer to the help documentation for the particular control.

The *KryptonButton* uses the states in the following way. If the control is disabled because the *Enabled* property is defined as *False* then it uses the *StateDisabled* values. When the control is enabled but the user is not interacting with it then it uses *StateNormal*. Once the mouse moves over the control area it uses *StateTracking* and if the user presses the mouse down whilst hovering over the control is uses *StatePressed*. By altering the appropriate state values you can easily achieve the look required for your application. Figure 4 shows an example of the appearance for each of the states for the *KryptonButton*.

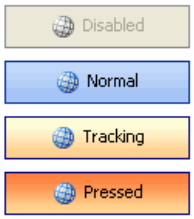


Figure 4 – KryptonButton states

Having the ability to alter each state is great for customization but can lead to slow development if you need to alter the same property for each of the states. For example, changing the border width is something you will probably want to be consistent across all the states.

Changing the border width for *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed* is tedious and prone to errors. This is where the extra *StateCommon* property comes into play. This extra property acts as a common base that all the other states use if they have not been given a value. In our example you just set the border width of *StateCommon* and all the states will inherit this value, unless you specifically set a value for one of the specific states.

Override
Some controls such as the *KryptonButton* need to alter the appearance of a state because of an external factor. For example, a button is capable of having the focus because the user can select it using the keyboard. In this case we need to give the user feedback so they can see it has the focus.

Properties that are applied to the current state based on an external factor such as the focus begin with the word *Override*. The *KryptonButton* has two such properties called *OverrideFocus* and *OverrideDefault*. In addition to focus the button can also be marked as the default button on the owning *Form*. The user needs feedback on this state because pressing the *Enter* key will cause the default button to be pressed.

The *Override* properties differ from *State* properties in that they do not inherit values from *StateCommon* and are applied to whatever the current state happens to be. Figure 5 shows two examples of the *KryptonButton* control. The top instance is the button in the normal state, the bottom instance shows the button also in normal state but with both the *OverrideFocus* and *OverrideDefault* applied.

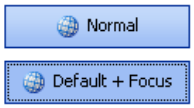


Figure 5 – Overrides applied to the normal state

Values

Many of the *Krypton* controls display information to the user. For example the *KryptonButton*, *KryptonLabel* and *KryptonHeader* controls all have the ability to show some text and/or an image as part of the appearance. This display content is always stored in a property called *Values*. *Content* always consists of two text values and an image that are grouped together. See Figure 1 for an example of the *Values* property for a *KryptonButton*.

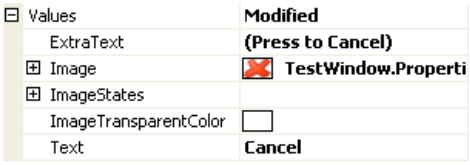


Figure 6 – KryptonButton values property

The exact name of the three properties varies depending on the context in which the content is being used. The *KryptonButton* calls them *Text*, *ExtraText, Image and ImageStates* whereas the *KryptonHeader* uses the names *Header*, *Description* and *Image*. Some controls have more than one content property. The *KryptonHeaderGroup* control has two value properties that contain content values, one of the primary header and another for the secondary header. Figure 7 shows the value properties for a *KryptonHeaderGroup*.

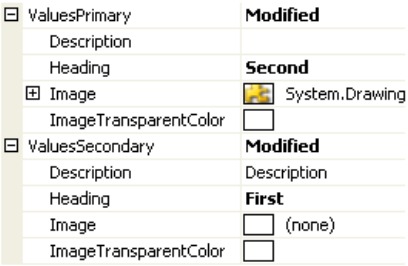


Figure 7 – Value properties for KryptonHeaderGroup

Background, Border and Content

Each *Krypton* control has a number of possible states that can each be customized in appearance. Rather than have a completely different set of customization properties for each control there are instead three standard sets that help to improve consistency. The standard sets of values are for the background, border and content.

Not every control will have all three sets as only the relevant ones are provided. So the *KryptonPanel* only has the background set as it does not have a border capability and does not display any content. The *KryptonLabel* has only content properties and so the background and border are not provided. All three are provided for the *KryptonButton* because it does have a background, a border and content values.

Some controls will have extra values in addition to those in the three standard sets as they have extra requirements that are control specific. But once you are familiar with how the three standard sets are used to customize all aspects of the control you will find it easy to deal with any of the *Krypton* controls. Each of the three sets of properties are explained in detail in the following section of the documentation.

Background

Most of the *Krypton* controls draw a background and so provide a set of properties that allow customization of the background appearance. Each control will provide the same set of properties for this task and so once you are familiar with how they operate you can use this knowledge when dealing with any of the controls. Figure 1 shows the standard set of properties from the *KryptonPanel* control.

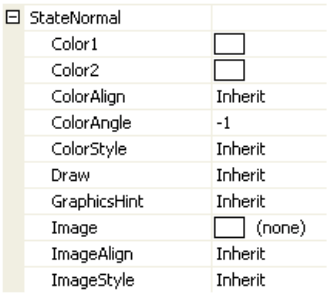


Figure 1 –Background display properties

Color Properties

The *ColorStyle* property determines how to make use of the *Color1*, *Color2*, *ColorAlign* and *ColorAngle* set properties. When *ColorStyle* is defined as *Solid* the background will be drawn using the *Color1* value and the other color properties are ignored as not relevant. Figure 2 shows some *KryptonPanel* instances with *Solid* defined.

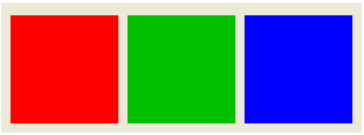


Figure 2 – ColorStyle = Solid

All other *ColorStyle* values such as *Linear* and *Rounded* allow a gradient effect between two colors, where *Color1* is the starting color and *Color2* is the ending color. Now the *ColorAngle* property becomes important as it determines the direction in which the transition occurs. Figure 3 shows the same *Linear* style applied with different angles.

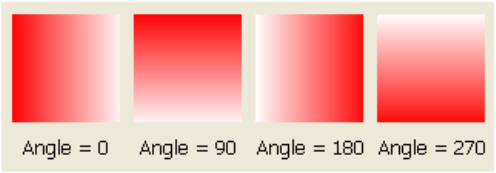


Figure 3 – ColorStyle = Linear

ColorAlign is used to decide how to calculate the starting and ending points for the gradient effect. By default it will always start at the top left of the control and ends at the bottom right of the control which gives appearance as seen already in figure 3. If instead you choose the *Form* level alignment then it will use the top left of the enclosing form and bottom right of the enclosing form as the total area of the gradient. Figure 4 shows four panel controls all with exactly the same *ColorAngle* of 45 degrees, *ColorAlign* of Form and *ColorStyle* of *Sigma*.

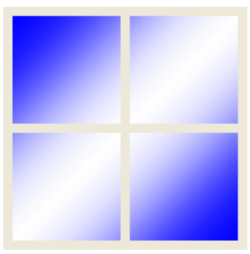


Figure 4 – ColorStyle =Sigma, ColorAlign = Form, ColorAngle = 45

Figure 5 shows a selection of other *ColorStyle* values not yet demonstrated. There are many other style options that you can explore in order to achieve the exact look and feel you need. Try using the ones beginning with *Glass* to achieve an effect similar to that of

Office 2007.

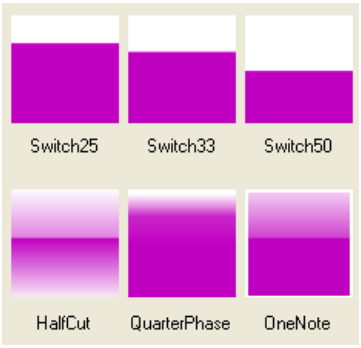


Figure 5 - Selection of some other ColorStyle options

Image Properties

The image is always drawn after the color has been drawn. This ensures that any alpha channel in the image will show through with the correct back color. Providing an *Image* is optional and the related *ImageAlign* and *ImageEffect* are only used when an *Image* has been specified. Figure 6 shows a *KryptonPanel* with an *Image* of a red cross with an alpha channel so that the *Color1* value of *Gold* shows through the transparent areas.

Figure 6 – Color1 = Gold, Image contains alpha channel

The *ImageAlign* property allows the image to be draw aligned against the *Form* and not just the control instance. Figure 7 shows the use of four *KryptonPanel* controls that are all assigned the same *Image*, *ImageAlign* and *ImageEffect*.

Figure 7 – ImageAlign = Form, ImageEffect = Stretch

The *ImageEffect* property describes how to draw the *Image* over the required area. Figure 6 shows the example where *Stretch* is used to ensure the *Image* exactly covers the entire area. Tiling options include *Tile*, *TileFlipX*, *TileFlipY* and *TileFlipXY* that repeat over and over again until the entire area is filled. Positioning options such as *TopLeft* and *BottomMiddle* allow the image to be placed relative to both the vertical and horizontal edges.

Other Properties

*Draw*is used to determine if the background of the control should be drawn at all. If you set this to be *False* then no background drawing will take place and so the control becomes transparent, instead showing through the container that parents the control.

GraphicsHint is more specialized and used to describe the level of quality used when drawing the background. This is generally only needed if you are using a *Krypton* control that has a border as well as a background and where the border itself is not being drawn but does provide rounding. For example, figure 8 shows a *KryptonGroup* that has a solid blue background and a rounded thick border of red.

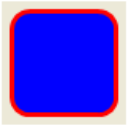


Figure 8 – KryptonGroup with rounded border

If we decide to remove the border because we only want the background to be visible but we still want the rounding effect then we set the borders *Draw*property to be false and the result can be seen in figure 9.

Figure 9 – Rounded border removed

In figure 8 you can see that the rounded edges are of a poor quality. This is where the *GraphicsHint* property is used to improve the smoothness of the drawing by setting the *AntiAlias* value. Figure 10 shows the result of changing the property.

Figure 10 – GraphicsHint = AntiAlias

Border
Border

The majority of *Krypton* controls allow a border to be drawn and so provide a set of properties that allow customization of the border appearance. Each control will provide the same set of properties for this task and so once you are familiar with how they operate you can use this knowledge when dealing with any of the controls. Figure 1 shows the standard set of properties from the *KryptonGroup* control.

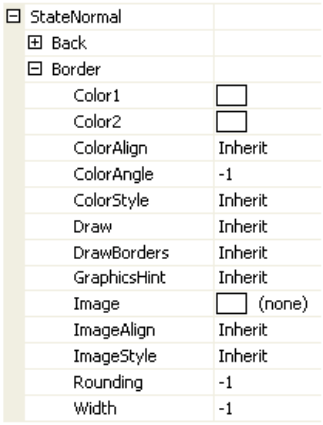


Figure 1 –Border display properties

Color Properties

The *ColorStyle* property determines how to make use of the *Color1*, *Color2*, *ColorAlign* and *ColorAngle* set properties. When *ColorStyle* is defined as *Solid* the border will be drawn using the *Color1* value and the other color properties are ignored as not relevant. Figure 2 shows some *KryptonGroup* instances with *Solid* defined and with a border *Width* of 7 pixels.

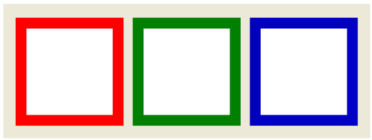


Figure 2 – ColorStyle = Solid, Width = 7

All other *ColorStyle* values such as *Linear* and *Rounded* allow a gradient effect between two colors, where *Color1* is the starting color and *Color2* is the ending color. Now the *ColorAngle* property becomes important as it determines the direction in which the transition occurs. Figure 3 shows the same *Linear* style and same starting and ending colors but with different angles.

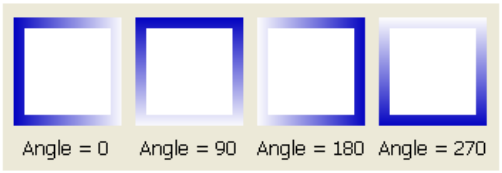


Figure 3 – ColorStyle = Linear

ColorAlign is used to decide how to calculate the starting and ending points for the gradient effect. By default it will always start at the top left of the control and ends at the bottom right of the control which gives appearance as seen already in figure 3. If instead you choose the *Form* level alignment then it will use the top left of the enclosing form and bottom right of the enclosing form as the total area of the gradient. Figure 4 shows four controls all with exactly the same *ColorAngle* of 45 degrees, *ColorAlign* of *Form* and *ColorStyle* of *Sigma*.

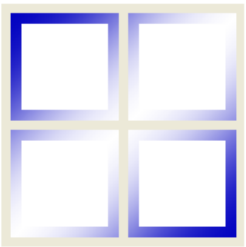


Figure 4 – ColorStyle =Sigma, ColorAlign = Form, ColorAngle = 45

Draw & DrawBorders Properties

Use the *DrawBorders* to specify the combination of four borders that should be allocated space. In the properties window you can use the drop down control that appears when pressing the property edit button to alter the setup. You can specify no border edges, just one border edge, two border edges, three border edges or all four. See figure 5 for just four examples of different border drawing settings.

Figure 5 – DrawBorders with a variety of options

As you expect, the *Draw* property is used to indicate if the border edges specified in *DrawBorders* should be drawn. It is important to note that if you use *False,* and so prevent drawing of the border edges, it will still allocate space for the borders and clip the background that is drawn underneath the border. You can see this in figure 6 where the same controls as figure 5 are used but with the *Draw* property set to *False*. If you no border space to be allocated on any edge and no borders to be drawn then you should set the *DrawBorders* property to none.

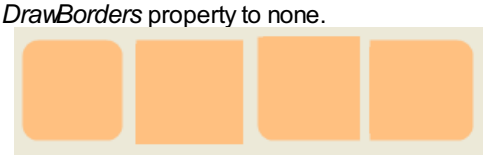


Figure 6 – Draw= False

Image Properties

The image is always drawn after the color has been drawn. This ensures that any alpha channel in the image will show through with the correct back color. Providing an *Image* is optional and the related *ImageAlign* and *ImageEffect* are only used when an *Image* has been specified. Figure 7 shows a *KryptonGroup* with an *Image* of a red cross with an alpha channel so that the *Color1* value of *Yellow* shows through the transparent areas.

Figure 7 – Color1 = Yellow, Image contains alpha channel

The *ImageAlign* property allows the image to be draw aligned against the *Form* and not just the control instance. Figure 8 shows the use of four *KryptonPanel* controls that are all assigned the same *Image*, *ImageAlign* and *ImageEffect*.

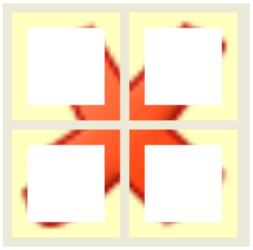


Figure 8 – ImageAlign = Form, ImageEffect = Stretch

The *ImageEffect* property describes how to draw the *Image* over the required area. Figure 6 shows the example where *Stretch* is used to ensure the *Image* exactly covers the entire area. Tiling options include *Tile*, *TileFlipX*, *TileFlipY* and *TileFlipXY* that repeat over and over again until the entire area is filled. Positioning options such as *TopLeft* and *BottomMiddle* allow the image to be placed relative to both the vertical and horizontal edges.

Width and Rounding

Width is used to determine the pixel width of the border. A value of -1 is used to indicate that the value should be inherited from the next highest level of settings. *Rounding* is used to decide how rounded the four corners of the border will be. Again a value of -1 will cause inheritance from the next highest level of settings. Figure 9 shows some examples of different *Rounding* values.

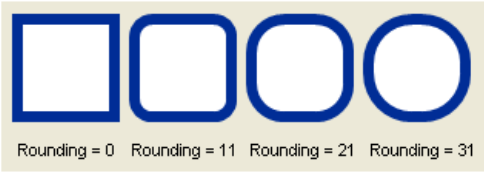


Figure 9 – Rounding property

Content

Content always consists of two text values and one set of image details. All of these are optional so you can choose to provide none, one, two or all three properties as needed. They are grouped together, for example the *KryptonButton* provides a property called *Values* that contains the values as child properties. See Figure 1 for an example of the property window for a *KryptonButton*.

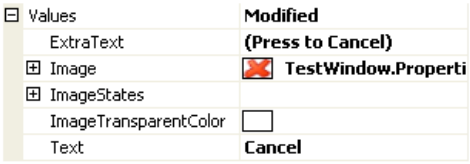


Figure 1 – KryptonButton values property

The exact name of the properties varies depending on the context in which the content is being used. The *KryptonButton* calls them *Text*, *ExtraText, Image, ImageStates* and *ImageTransparentColor* whereas the *KryptonHeader* uses the names *Header and Description* instead of *Text* and *ExtraText*. We will call them *ShortText*, *LongText*, *Image, ImageStates* and *ImageTransparentColor* for the duration of this section.

Any control that provides content values will also allow you to specify how the content is displayed for each of the control states. Figure 2 shows the set of properties available for controlling the display of the content values.

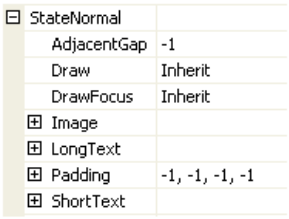


Figure 2 – Content display settings

Content Properties

AdjacentGap controls the pixel spacing between the different content values. Use this to change the distance between the *ShortText*, *LongText* and *Image*. Figure 3 shows a *KryptonLabel* with values of 1, 5 and 20 respectively.

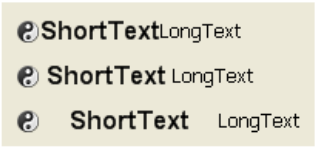


Figure 3 – AdjacentGap property

*Draw*is used to determine if any of the content should be displayed. When defined as *False* the content is ignored completely. *DrawFocus* indicates if a focus rectangle should be drawn around the edge of the content area. Figure 4 shows a *KryptonLabel* control with the *DrawFocus* defined as *False* and *True* respectively.

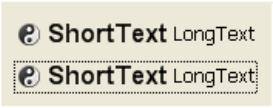


Figure 4 – DrawFocus property

Padding specifies how far to inset the content from the outside of the control. You can provide a value that is the same for all the sides or individual values for each of the edges. Figure 5 uses a *KryptonButton* control that is auto sized to demonstrate how padding values affect the size and drawing of the control.

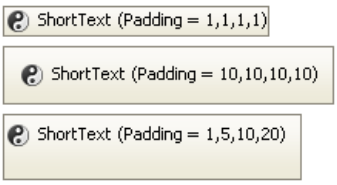


Figure 5 – Padding property

The remaining three properties are *Image*, *ShortText* and *LongText* that contain settings that affect just the corresponding parts of the content.

Positioning

Each of the three content parts has a property for controlling the horizontal alignment and another for the vertical alignment. These are called *ImageH*, *ImageV* for the *Image* and *TextH*, *TextV* for the *ShortText* and *LongText*. All of them have the same possible set of values *Inherit*, *Near*, *Center* and *Far*.

Inherit is used to indicate that the value of the property should be determined by looking higher up the inheritance chain. On a left to right system the *Near*, *Center* and *Far* are interpreted as Left/Top, Middle and Right/Bottom. Right to left systems will interpret this in the reverse order of Right/Bottom, Middle and Left/Top.

Positioning is easiest to understand by using examples. In figure 6 a sequence of images shows the *ShortText* positioned inside a fixed size *KryptonButton*. In each case the actual display text is the horizontal and vertical alignment values.

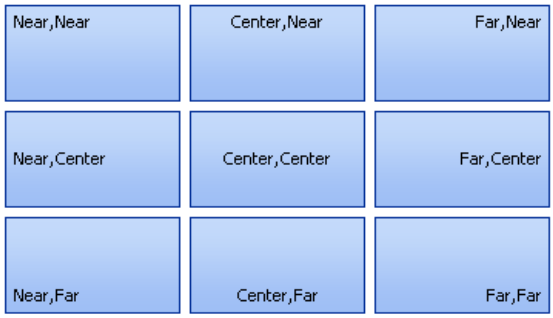


Figure 6 – TextH, TextV positioning of ShortText only

If you position more than one of the content elements in the same place then they simply concatenate together in the order of *Image*, *ShortText* and *LongText*. So if all three were positioned at (*Far*, *Far)* then you would get the situation shown in figure 7. Note that the *Image* is positioned first and so takes the furthest position horizontally. Then the *ShortText* is added and finally the *LongText*.

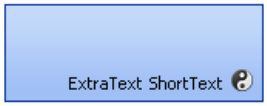


Figure 7 – Far, Far positioning of all content

You can achieve most affects you will need by using these settings. For example, if you would like to see the *LongText* at the bottom with the *ShortText* above and finally the *Image* at the top then you use the (*Center*,*Top*) for the *Image* then define (*Center*,*Center*) for the *ShortText* and finally (*Center*,*Bottom*) for the *LongText*. This would achieve the appearance of figure 8.

Figure 8 – Content in centered for spread out vertically

If you need to squeeze the content more closely together then you have two options. The preferred method is to auto size the control so that it automatically resizes to be just big enough to hold the content. You could also add a larger *Padding* value so that the spare space is reduced forcing the content closer together. Figure 9 shows the auto sizing approach followed by the *Padding* approach where it has been defined as 22 pixels of *padding* for each edge. The last picture in figure 9 shows both the auto sizing and *Padding* applied to the same control.

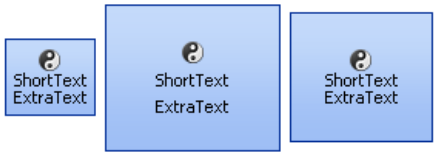


Figure 9 – AutoSize, Padding and both

Image Properties

As you can see from Figure 10 the *Image* property has five child properties of which the *ImageH* and *ImageV* have already been described above. The *Effect* property allows a color transition to be made at the time of drawing the actual image to the screen.

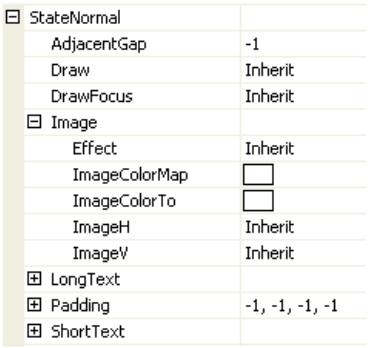


Figure 10 – Image child properties

The most commonly used enumeration values are the *Normal*, which just shows the image without any change, and *Disabled* which shows the image is a washed out appearance suitable for a disabled control. You can lighten or darken the image by using one of the *Light*, *LightLight*, *Dark* and *DarkDark* options. There are four options for performing gray scaling. *GrayScale* will convert all colors into black and white whilst *GrayScaleRed*, *GrayScaleGreen* and *GrayScaleBlue* convert all except the nominated color into gray scale. See figure 11 to see all the enumeration options applied.

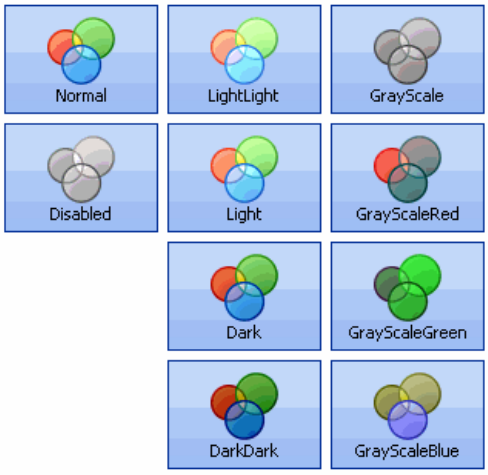


Figure 11 – Image Effect enumeration

When the image is being painted you can remap one of the image colors to an alternate. Use the *ImageColorMap* to specify the color you want to alter and use *ImageColorTo* to define the new output color. This is useful if you have just a single color image and want to alter the display color based on the tracking or pressed state. Figure 12 shows a black cross image in three colors. The left most image is the original Image without any color re mapping applied. In the second image the *ImageColorMap = Black* and *ImageColorTo = Red*. Last is the blue image which also has *ImageColorMap = Black* but *ImageColorTo = Blue.*

Figure 12 - Original image, ColorTo = Red, ColorTo = Blue

ShortText & LongText Properties

Figure 11 shows the full list of extra properties that are available for both the *ShortText* and *LongText*. The *TextV* and *TextH*

properties have already been described above in the section about positioning.

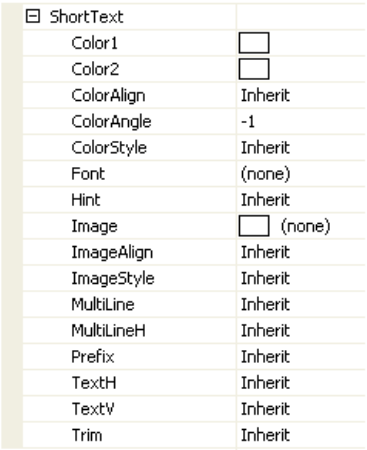


Figure 11 – ShortText child properties

The *ColorStyle* property determines how to make use of the *Color1*, *Color2*, *ColorAlign* and *ColorAngle* set properties. When *ColorStyle* is defined as *Solid* the text will be drawn using the *Color1* value and the other color properties are ignored as not relevant. Figure 12 shows some *KryptonLabel* instances with *Solid* defined.

Figure 12 – ColorStyle = Solid

All other *ColorStyle* values such as *Linear* and *Rounded* allow a gradient effect between two colors, where *Color1* is the starting color and *Color2* is the ending color. Now the *ColorAngle* property becomes important as it determines the direction in which the transition occurs. Figure 13 shows the same *Linear* style and same starting and ending colors but with different angles.

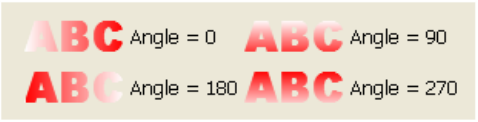


Figure 13 – ColorStyle = Linear

ColorAlign is used to decide how to calculate the starting and ending points for the gradient effect. By default it will always start at the top left of the control and ends at the bottom right of the control which gives an appearance as seen already in figure 13. If instead you choose the *Form* level alignment then it will use the top left of the enclosing form and bottom right of the enclosing form as the total area of the gradient. Figure 14 shows four label controls all with exactly the same *ColorAngle* of 45 degrees, *ColorAlign* of *Form* and *ColorStyle* of Sigma.

Figure 14 – ColorStyle =Sigma, ColorAlign = Form, ColorAngle = 45

Font and *Hint* are used to define the text font used when drawing and the quality of the font rendering. Figure 15 shows the *Font* modified between the first and second instances. The third instance is exactly the same as the second but with the *Hint* defined as *AntiAlias*.

Figure 15 – Font and Hint properties

Rather than just use colors you can also draw an *Image* over the text. Note that the *Image* is always drawn after the color has been painted so that images that contain alpha channels will show through the defined color effect.

The *ImageAlign* property allows the image to be draw aligned against the *Form* and not just the control instance. Figure 16 shows the use of many *KryptonLabel* controls that are all assigned the same *Image*, *ImageAlign* and *ImageEffect*.

Figure 16 – ImageAlign = Form, ImageEffect = Stretch

The *ImageEffect* property describes how to draw the Image over the required area. Figure 16 shows the example where *Stretch* is used to ensure the *Image* exactly covers the entire area. Tiling options include *Tile*, *TileFlipX*, *TileFlipY* and *TileFlipXY* that repeat over and over again until the entire area is filled. Positioning options such as *TopLeft* and *BottomMiddle* allow the image to be placed relative to both the vertical and horizontal edges.

Prefix has three possible values that determine how to process the ampersand character inside the text string. *None* specifies that no pre-processing of ampersand characters take place so the output is the exact string provided. *Hide* will process ampersand characters but will not underline the output text. *Show*will process ampersand characters and place an underline under the next character. Figure 17 shows the each of the three options.

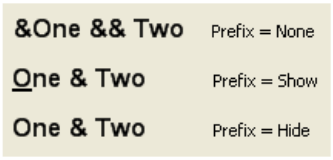


Figure 17 – Prefix options

Each *ShortText* and *LongText* can contain carriage returns and so span multiple lines of text. This is the default setting for *MultiLine* but if you prefer to have all the text always on a single line then just set the property to *False*. When text spans multiple lines then you can use the *MultiLineH* property to define how the lines are aligned. Figure 18 shows an example of text containing a carriage return being forced to show on a single line and then an example of the three alignment options.

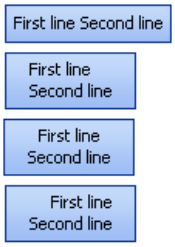


Figure 18 – MultiLine and MultiLineH properties

Finally the *Trim* property is used to specify how to handle text when there is not enough room to display it all. If you would like the text to disappear then use the *Hide* option. Alternatively you can use the *Character* or *Word* options to remove the excess characters or words that that will not fit. If you would like the user to be made aware that truncation has taken place then use one of the ellipses options *EllipsesCharacter*, *EllipsesWord* or *EllipsesPath*. Figure 19 shows the various options in action.

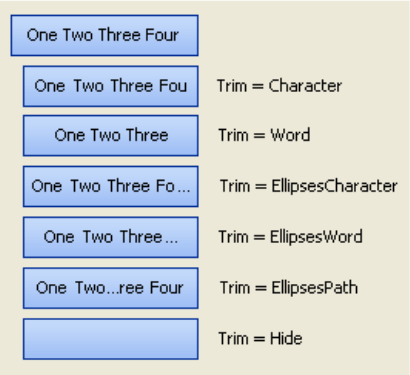


Figure 19 - Trim options

ButtonSpec

Some *Krypton* controls allow buttons to be specified for display within the *Krypton* control itself. In the case of the *KryptonHeader* and *KryptonHeaderGroup* they are displayed in the header portions of the control. In order to support this capability they expose a collection property called *ButtonSpecs.* This collection can contain any number of *ButtonSpec* (Button Specification) instances that describe the button that is required for display. Figure 1 below shows the *ButtonSpecs* property as it is exposed by the *KryptonHeader* control.

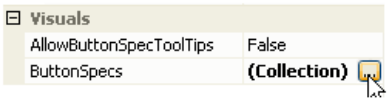


Figure 1 - ButtonSpecs property on KryptonHeader

ButtonSpec Instances
On clicking the property button the standard collection editor is displayed as a modal dialog box. You can then use the collection editor to create and remove individual *ButtonSpec* instances. On the right side of the collection editor you will see the properties of the currently selected *ButtonSpec* instance. Figure 2 shows the properties that are exposed for a *ButtonSpec* instance.

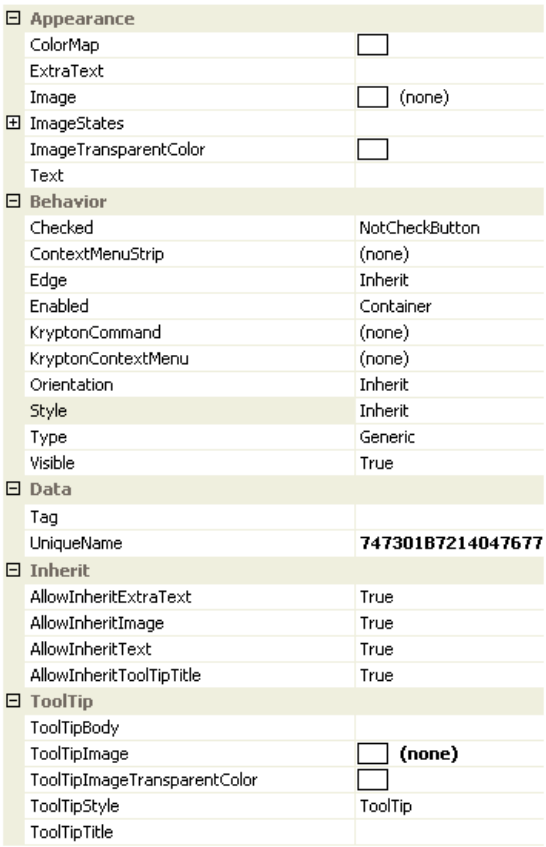


Figure 2 - ButtonSpec properties in collection editor

To describe the function of the different properties a *KryptonHeader* control will be used. Figure 3 shows a simple *KryptonHeader* instance that has been created and currently does not have any *ButtonSpec* instances added.

Figure 3 - KryptonHeader with no buttons

Text, ExtraText, Image, ImageStates and ImageTransparentColor
The *ButtonSpec* exposes the same value properties as a *KryptonButton* instance. You can assign text for display by setting the *Text* and *ExtraText* properties. To show an image you use the *Image* property and if you need per-state images then you can use the properties available as children of *ImageStates*. If you need to specify which color in the *Image* should be transparent then use the *ImageTransparentColor* property. Figure 4 shows a red cross assigned to the *Image* property and all other properties of the

ButtonSpec left as default values. At the top of figure 4 is the output when the mouse is not interacting with the button, the second picture shows the mouse tracking over the button and finally when the mouse is pressed down.

Figure 4 - Button in normal, tracking and pressed states

If you need to show different images for each of the button states then instead of using the *Image* property you would assign the different images to the child properties of *ImageStates*. You would not usually define all of the *Text*, *ExtraText* and *Image* properties together because of the excessive amount of space they would occupy but Figure 5 shows that it is possible.

Figure 5 - ButtonSpec with Image, Text & ExtraText defined

ToolTipTitle, ToolTipBody, ToolTipImage, ToolTipImageTransparentColor and ToolTipStyle
These properties are used to specify the tool tip details for display when the user hovers over the button spec instance. Use the *ToolTipTitle* and *ToolTipBody* properties to define two text strings for display. To associate an image with the tool tip you should assign it to *ToolTipImage* and use the *ToolTipImageTransparentColor* for specifying a color in the image that should be treated as transparent. For example, many bitmaps will use magenta as a color for the background area that should become transparent when the bitmap is drawn, in that case assign Color.Magenta to the *ToolTipImageTransparentColor* property.

The default value for the *ToolTipStyle* is *LabelStyle.ToolTip* and will cause the image and title text to be shown at the top of the tool tip area and the body text to be shown below. Alternatively you could change the style to *LabelStyle.SuperTip* in which case the title text is shown in bold at the top of the tool tip area, the image in shown below with the title also below and to the right of the image.

Note that by default the *Krypton* controls that allow the definition of button specification do not show tool tips for them. You must therefore find the appropriate property such as *AllowButtonSpecTooltips* and set it to *True* before tool tips will be displayed.

Style
By default the button will be displayed using the *ButtonSpec* button style appearance. Note that the *Standalone* button style is intended for use with standard buttons within the main application, *LowProfile* is likewise intended for standalone buttons that need a lower profile. *ButtonSpec,* as the name indicates, is intended for use with button specification scenarios and has a much smaller padding between the button contents and the border. This is because space is very constrained in places that a *ButtonSpec* is used and so the button needs to be kept compact.
If you prefer to change the default *ButtonSpec* button style then just modify the *Style* property. Figure 6 shows the appearance when the *Style* is altered to *Standalone*, where a background and border are shown even when the mouse is not over the button.

Figure 6 - Style = Button1

Edge
All the examples so far have shown the button placed on the right hand side of the example *KryptonHeader*. This is the default, *Far*, position but you can reverse this to have the button shown *Near*. Figure 7 shows the *Edge* property modified to *Near*.

Figure 7 - Edge = Near

Enabled
The *Enabled* property has three possible values. If you specify *Enabled* as *True* then the button will be enabled as long as the control itself is also enabled. Remember that if the *Krypton* control itself has been disabled then each *ButtonSpec* must be disabled because the user has been prevented from interacting with the control. If you specify *Enabled* as *False* then the button is always displayed as disabled as can seen in figure 8.

The third possible *Enabled* property value is *Container*. The need for three different enabled options is not apparent on a simple control like a *KryptonHeader* but is required for more complex controls like the *KryptonNavigator*. The *Container* option acts the same as the *True* setting for simple controls where the whole control is the container. If the control (which is also the container) is enabled then so is the *ButtonSpec,* if the control is disabled then so is the ButtonSpec. Hence the *True* and *Container* options act in exactly the same way. For the *KryptonNavigator*, and other complex controls, the container is not the whole control and so the semantics are altered. Refer to the documentation for the complex control for control specific details.

Figure 8 - Enabled = False

Visible
By default the *Visible* property is defined as *True* so that the *ButtonSpec* is shown. Alter this to *False* to remove it from the display.

Checked
If you need your *ButtonSpec* to act like a toggle button instead of the default push button then you need to alter the *Checked* property. The default value of this property is *NotCheckButton* and so it is always drawn as a standard push button. If you alter the property to either *Checked* or *Unchecked* then pressing the button with cause it to toggle between those two values. Figure 9 shows the *Checked* property defined as *Checked.*

Figure 9 - Checked = Checked

Type
By default the *Type* property for a new *ButtonSpec* is defined as *Generic*. This means the button has no predefined meaning and you must supply the image and text that you would like displayed. There are however some common types of button and you can indicate that your button should be of that predefined type. In these case the image and text are inherited from the palette if you do not provide them yourself in the *ButtonSpec* properties. Figure 10 shows four of the predefined types of *Previous*, *Next*, *Context* and *Close* and the images that are inherited from the palette. There are other predefined types including *ArrowUp*, *ArrowDown*, *ArrowLeft, ArrowRight, FormClose, FormMin, FormMax* and *FormRestore*.
Because the image and text values are inherited for predefined types you need a mechanism to specify no image or no text. If you leave the *Image* property blank then the image is pulled in from the palette. But what if you really do not want any image? Then you can assign *False* to the *AllowImageInherit* property. Likewise you can use the *AllowTextInherit* and *AllowExtraTextInherit* to prevent the text values from being inherited.

Figure 10 - Previous, Next, Context and Close types

Orientation
What happens to the display of the buttons when the owning control has a different orientation? Figure 11 shows that by default the buttons are also rotated in the same way as the owning control. However this might not always be appropriate for your application as the *Previous* and *Next* predefined types no longer point left and right respectively.

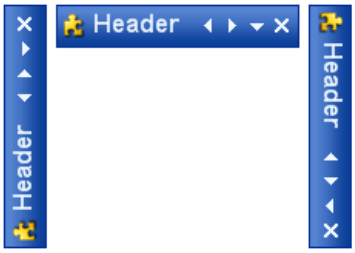


Figure 11 - Orientation = Auto

This is purpose of the *Orientation* property on the *ButtonSpec*. The default value of Auto will cause the button to be rotated in line with the owning control as seen above. You can however fix the orientation of the *ButtonSpec* to a particular value. Figure 12 shows the *Orientation* property defined as *FixedTop* which now makes more sense to the user.

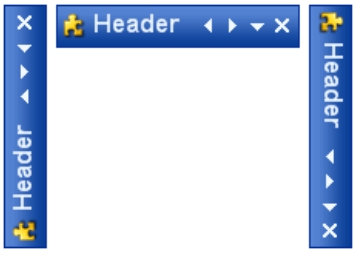


Figure 12 - Orientation = FixedTop

ColorMap
If you look at Figure 13 you will see that four predefined types have been used on to *KryptonHeader* controls, the top header is using a

Header1 style and the bottom header *Header2* style. Although the builtin palette has only a single black image for each predefined type the headers are still showing the images colored appropriately for each header style. This is where the *ColorMap* property comes into play.
When the *ColorMap* property is defined as a color then the button will remap that specified color to the color used for drawing the main content text. As can be seen in figure 13 the top header is using white for the main content text and the second header is used black, hence the buttons have been re mapped from black to the appropriate color. The *ColorMap* value is inherited from the palette and so when you specify a predefined type of *Close* the palette will automatically map the black that is used in the *Close* image. When you specify your own image or use the *Generic* type then you would need to set the *ColorMap* manually to get the functionality.

Figure 13 - Types with default color re mapping

It is not just the image that is re mapped but also the text as well. Figure 14 shows a *ButtonSpec* that is defined as the *Close* type and where additional *Text* has been provided as well. In the top picture you can see that text is also in white to match the main content color. The second picture shows the main text changed to green and also how the re mapping is continuing to work.

Figure 14 - Color re mapping

Mnemonics
As with the standard *KryptonButton* you can use the ampersand character to place an underline with the following character. If the owning control that contains the *ButtonSpec* definition processes a mnemonic character then it will check each *ButtonSpec* to see if the mnemonic matches it. If so then the *Click* event for the ButtonSpec will be fired. Check that the owning control has the *UseMnemonic* property defined as *True*.

ContextMenuStrip
If you assign a *ContextMenuStrip* reference to the property of the same name then clicking the button will cause the context menu to be displayed. Note that this property is only used if the *KryptonContextMenu* property is not defined.

KryptonContextMenu
Clicking the button will cause this context menu to be shown below the button on the screen. This property takes precedence over the *ContextMenuStrip* property as defined above. Only if this property is *null* will the *ContextMenuStrip* be tested and used instead. Use of the context menu properties is a quick way to add context functionality to a *KryptonHeader*, *KryptonHeaderGroup, KryptonRibbon* or *KryptonNavigator* control.
UniqueName
As the name suggests, this field allows the developer to assign a unique name to the button specification. By default each new instance will be assigned a new GUID generated from the operating system to ensure uniqueness. As these are not easy to remember it is recommended you alter the property to a more meaningful value if you intend to make use of the property

Tag
Use this to associate your application specific information with the button spec instance.

Click Event
The standard *Click* event is fired whenever the user presses the button represented by the button specification.

Palettes

Using palettes allows you to quickly alter the look and feel of your application. By altering the global palette setting you can cause all the *Krypton* controls to update to the new appearance in one action. Alternatively you can modify the palette to be used on a per toolkit control basis in order to use different palettes for different controls. There are three sources of palettes available to you as a toolkit developer.

Built in palettes

The *Krypton Toolkit* comes with built in palettes that are embedded in the library itself. For example the *Office 2007 - Blue* and *Professional - Office 2003* palettes are provided out of the box. You can use the *KryptonManager* component to alter the global palette to one of the built in choices. Figure 1 shows the global palette being altered to a built in palette.

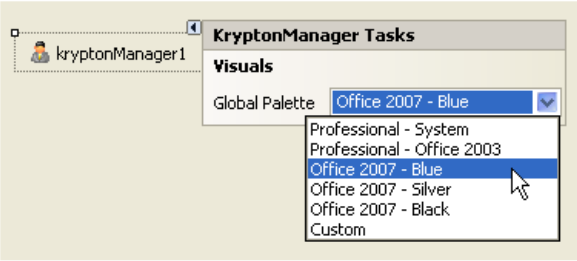


Figure 1 - KryptonManager smart tag

Import a palette definition

Your second option is to add a *KryptonPalette* component to the application and then import palette settings from a palette definition file. Palette definition files are just XML documents that store all the settings and images needed to recreate a palette. All you need to do is drop a new *KryptonPalette* onto your form and then click the smart tag.
The smart tag contains options to *Export* and *Import* palette definitions. Figure 2 shows the smart tag for the *KryptonPalette*.

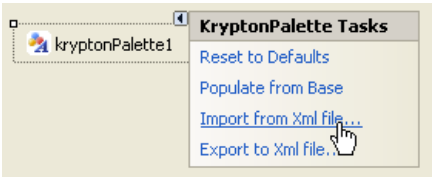


Figure 2 - KryptonPalette smart tag

Now you need to alter the *KryptonManager* property called *GlobalPalette* so that the *KryptonPalette* component is used instead of one of the built in options. Figure 3 shows the property being altered at design time.

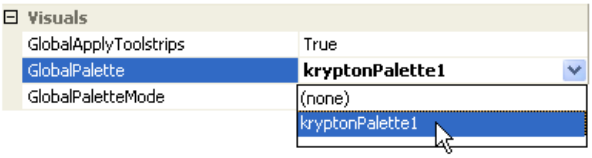


Figure 3 - KryptonManager GlobalPalette property

Note that exporting a palette will only save values that are not the default. So if you have only changed a few values from the default then the exported XML file will be very small and compact. When importing a palette the only values that will be changed in the palette are those than are read in from the XML file. If you have a new instance of a *KryptonPalette* component then all the values will be defaulted and importing a palette will achieve the expected changes.
If you expect that some of the palette values are not defaulted and might not be changed by the incoming XML file then you should perform a *Reset to Defaults* before the import; figure 2 shows the reset task on the smart tag. This resets all palette properties back to the default values they would have when the component was first created.

Populate from base

Another option available from the palette smart is called *Populate from base*. This option will update all the values in the palette with the values that are available from the base palette. So if the base palette is the built in *Professional - Red* then the values will consist of the red gradient colors the base exposes. Note that all styles prefixed *Custom* will not be imported as these styles are always local to a particular palette.

Creating a palette definition

You can create a new palette definition instead of importing an existing definition. You could do this by altering the properties of the *KryptonPalette* using the properties window at design time. This is however quite difficult as you cannot see the changes taking effect immediately unless you have every type of control on your form at the same time. Instead you should use the *Palette Designer* utility that can be started from the *Krypton Explorer* program. There should be a link to the explorer application placed on your desktop

during the install process.

Use the *Palette Designer* to modify the palette properties and see the changes immediately. Once you are finished you can save the settings to a palette definition file. Then just use the smart tag on the *KryptonPalette* to access the *Import* option and select the file just saved from the *Palette Designer*. Figure 4 shows the Palette Designer.

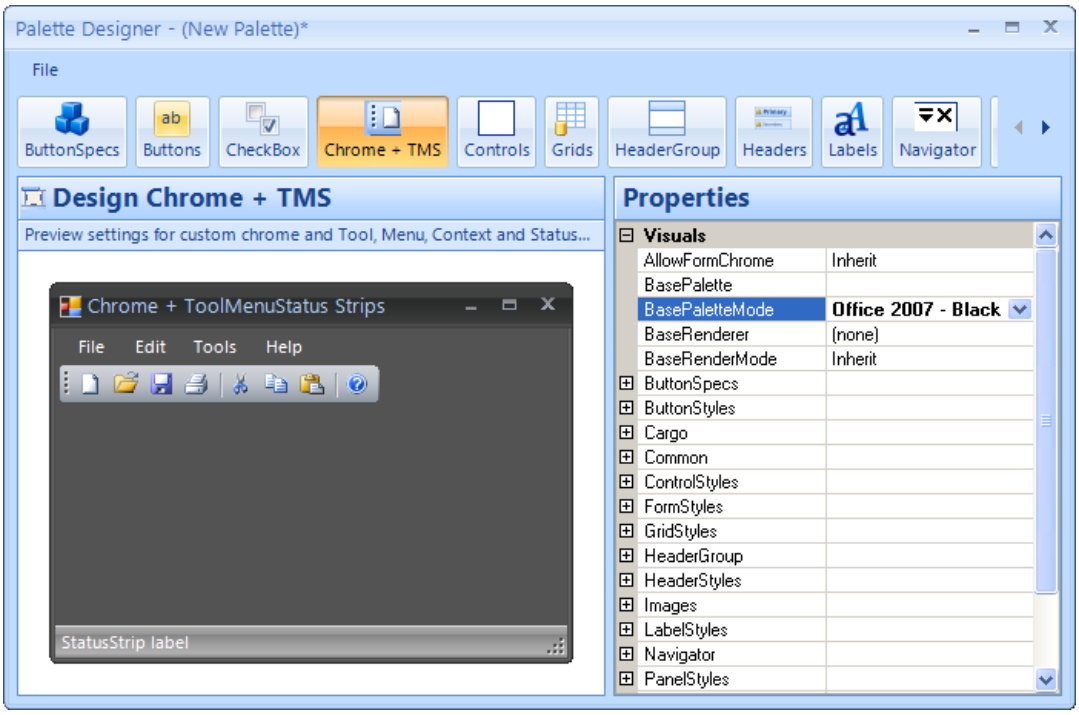


Figure 4 - Palette Designer

Upgrading Palette Definitions

When moving from one version of *Krypton* to another there are sometimes changes made to the palette XML format. When this happens you can use the *Palette Upgrade Tool* to convert palette XML files to the latest file format. This utility can be started from the *Krypton Explorer* program, a shortcut for which should be on your desktop after installation. Figure 5 shows the tool in action.

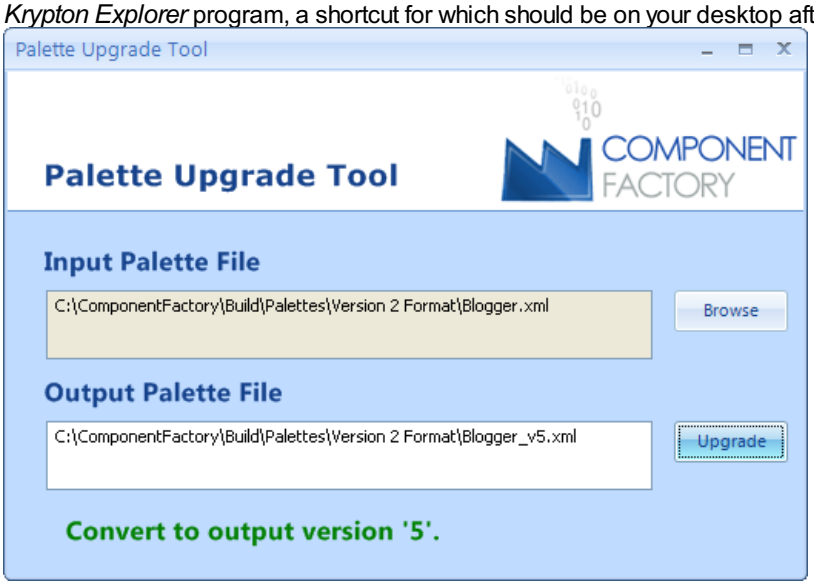


Figure 5 - Palette Upgrade Tool

You can make use of the tool to upgrade *Krypton Palette* component instances. Before installing a new version of *Krypton* you can use the *Export* option on the palette in order to create a palette definition file. Then perform a *Reset* on the palette component so it is defaulted for all its values. Once the new version of *Krypton* has been installed you use the *Palette Upgrade Tool* to update the XML file and then *Import* it into the palette component. This is easier than trying to fix the compile time errors that would result from palette properties being removed or renamed.

Toolkit

Learn more about each of the individual controls and components that appear in the toolbox.

Controls

- KryptonBorderEdge
- KryptonBreadCrumb
- KryptonButton
- KryptonCheckBox
- KryptonCheckButton
- KryptonCheckedListBox
- KryptonColorButton
- KryptonComboBox
- KryptonDataGridView
- KryptonDateTimePicker
- KryptonDomainUpDown
- KryptonDropDown
- KryptonGroup
- KryptonComboBox
- KryptonHeader
- KryptonHeaderGroup
- KryptonLabel
- KryptonLinkLabel
- KryptonListBox
- KryptonMaskedTextBox
- KryptonMonthCalendar
- KryptonNumericUpDown
- KryptonPanel
- KryptonRadioButton
- KryptonRichTextBox
- KryptonSeparator
- KryptonSplitContainer
- KryptonTextBox
- KryptonTrackBar
- KryptonTreeView
- KryptonWrapLabel

Components

- KryptonCheckSet
- KryptonCommand
- KryptonContextMenu
- KryptonPalette
- KryptonInputBox
- KryptonManager
- KryptonMessageBox
- KryptonTaskDialog

Forms

- KryptonForm

KryptonBorderEdge

Use the *KryptonBorderEdge* control when you need to display a vertical or horizontal graphic that has a border appearance. For example, you could place a vertical *KryptonBorderEdge* instance inside a *KryptonGroup* in order to split the group into two visual areas. Using this control would ensure the appearance of the vertical graphic is consistent with the appearance of the *KryptonGroup* border. They would have the same colors and thickness etc.

Appearance

The *BorderStyle* property is used to define the styling required for the appearance of the *KryptonBorderEdge* control. The default value of *Control - Client* is the same as the default border style of the *KryptonGroup*. You can alter this property to any of the many border style enumeration values so that the appearance matches whatever control you are using.

You can use the *Orientation* property to specify either a vertical or horizontal. By default the control has an *AutoSize* value of *True* that will cause one of the control dimensions to be automatically calculated. When in the vertical orientation the width is automatically calculated and when horizontal the height is auto calculated. If you need to specify both the width and height then set the *AutoSize* to be *False*. See figure 1 for examples of both orientations.

Figure 1 – Width = 5, Orientation = Horizontal + Vertical

Two States

Only two possible states of *Disabled* and *Normal* are used by the border edge control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Examples of Appearance

Figure 2 shows the appearance of the border edge for the *Control - Client* style in the vertical and horizontal orientations. The left side shows the controls enabled and the right side shows the disabled appearance.

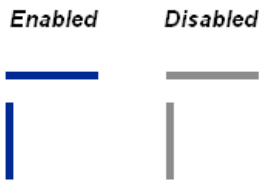


Figure 2 – Example appearance

KryptonBreadCrumb

Use the *KryptonBreadCrumb* control to navigate around a tree hierarchy of options. The control displays a trail leading from the root to the currently selected location. This is useful in allowing the user to quickly navigate to a previous node in the path that leads to the current location. It also has the advantage of only taking up a small area of the screen, unlike a traditional tree control. Use the *RootItem* property to define the hierarchy and the *SelectedItem* property to define the current selection.

Appearance

Use the *ControlBackStyle* and *ControlBorderStyle* properties to define the styling required for control and the *CrumbButtonStyle* to define the appearance of each individual path entry. You can see in figure 1 the default appearance using the *Office 2007 - Blue* builtin palette.

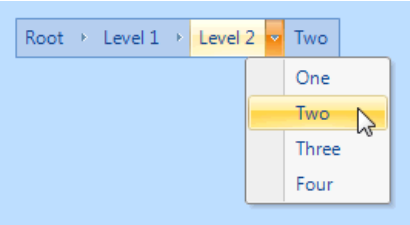


Figure 1 – Default Appearance

Four States

The visual elements can be in one of four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. The overall control can only be in the *Normal* or *Disabled* states but the individual crumbs, represented by drop down buttons, can also be in *Tracking* and *Pressed* states when interacting with the mouse.

In order to customize the appearance in each of the four states you can use the properties *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed*. Each of these properties allows you to modify the background, border and content characteristics of the drop down buttons as well as the border and background of the overall control.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

DropDownNavigation

By default this is defined as *True* and gives each individual bread crumb item a drop down button. When pressed the drop down button presents a context menu with a list of all available child items. On selecting one of the child items in the menu that child then becomes the newly selected item. Note that if a bread crumb item does not have any child items defined then it will not show a drop down button, even if this property is defined as *True*.

RootItem

This is the root item that starts the hierarchy of possible items. Each items has a *ShortText*, *LongText* and *Image* set of properties that are used to define the appearance of the item. Each item also has an *Items* property that allows child items to be attached. You can modify this hierarchy at design time to specify a fixed structure or alternatively modify the structure at runtime in response to application events.

SelectedItem

Use the *SelectedItem* to define the currently defined path in the control. Note that the reference **MUST** be one of the items inside the *RootItem* hierarchy. If you assign *NULL* then the control will be empty of any bread crumb displayed. If you assign the *RootItem* value then you will see a single bread crumb entry that represents the root of the available hierarchy. Otherwise you can assign one of the entries from the *RootItem* structure and it will show the entire path from the root to the provided reference.

ButtonSpecs

You can add buttons to the bread control control by modifying the *ButtonSpecs* collection exposed by the *KryptonBreadCrumb*. Each *ButtonSpec* entry in the collection describes a single button for display. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances. See the ButtonSpec section for more details. Figure 2 shows an example of a *KryptonBreadCrumb* with buttons.

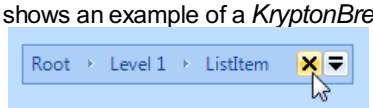


Figure 2 – KryptonBreadCrumb with ButtonSpecs

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

UseMnemonic

When this property is defined as *True* for the *KryptonBreadCrumb* it will check each of the *ButtonSpec* instances in the *ButtonSpecs* collection. If the *Text* or *ExtraText* in a *ButtonSpec* matches the incoming mnemonic then the *Click* event for the *ButtonSpec* will be fired.

KryptonButton

Use the *KryptonButton* control when you need button functionality combined with the styling features of the *Krypton Toolkit*. This control allows the user to initiate an action by pressing the button in order to generate a *Click* event. Place code in the *Click* event handler to perform the required action. The content of the control is contained in the *Values* property. You can define *Text*, *ExtraText* and *Image* details within the *Values* property.

Appearance

Use the *ButtonStyle* property to define the top level styling required for the appearance of the *KryptonButton* control. The default value of *Standalone* gives a solid appearance that should be appropriate for use in most circumstances. Alternatively you can use the *LowProfile* setting for situations where you need a transparent border and background. See figures 2 and 3 below for examples of the visual difference. A third button style *ButtonSpec* is used by default for button specifications that are placed inside *KryptonHeader* and *KryptonHeaderGroup* controls. Buttons that appear on the caption area of a *KryptonForm* use the *Form* style. There are also custom styles that can be defined via a *KryptonPalette* for situations where you need to create variations on the styles already provided. Custom styles are named simply *Custom1*, *Custom2* and *Custom3*.

You can use the *Orientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 1 for examples.

Figure 1 – Orientation Property

Four States

The button can be in one of four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. If the control has been disabled because the *Enabled* property is defined as *False* then the button will be in the *Disabled* state.

When enabled the button will be in the *Normal* state until the user moves the mouse over the button at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the control then it enters the *Pressed* state.

In order to customize the appearance of the control in each of the four states you can use the properties *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed*. Each of these properties allows you to modify the background, border and content characteristics.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the border of the button to be 3 pixels wide with a rounding of 2 pixels and always red. Without the *StateCommon* property you would need to update the same three border settings in each of *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed* properties. Instead you can define the three border settings in just *StateCommon* and know they will be used whichever of the four appearance states the button happens to be using.

State Overrides

There are two additional properties called *OverrideDefault* and *OverrideFocus* that are involved in altering the appearance of the control. Notice that these start with the *Override* prefix instead of the usual *State*. This is because they do not relate to a specific control state such as *Normal* or *Tracking*. Instead they can be applied to any of the four possible states and are used to override the appearance that would otherwise be shown.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied. This can occur when the control is in *Normal*, *Tracking* or *Pressed* states. By default the override only alters the appearance so that a focus rectangle is drawn around the content so that the user can see that the control currently has the focus.

Default Button Override

Each *Form* has the concept of a default button that is clicked when the user presses the *Enter* key. The *KryptonButton* also supports the default button mechanism and *OverrideDefault* settings are applied to the appearance whenever the control has been designated as the default.

Examples of Appearance

Figure 2 shows the appearance when the default *ButtonStyle* of *Standalone* is used. You can see that all the different states are

drawn as solid controls. The third button down shows how the default button and focus overrides have affected the appearance of the *Normal* state above it.

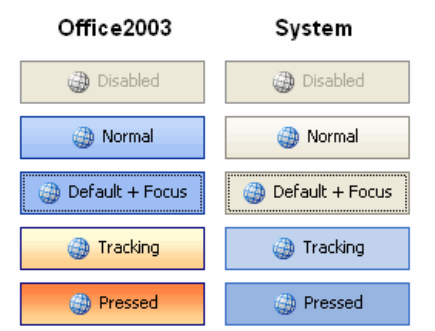


Figure 2 – ButtonStyle = Standalone

Figure 3 shows exactly the same states but this time the *ButtonStyle* is *LowProfile*. This time the background is transparent in the *Disabled* and *Normal* states. This button style is intended for use when you need to overlay a button onto a background and you want the button to be low profile and only light up when the user is interacting with it.

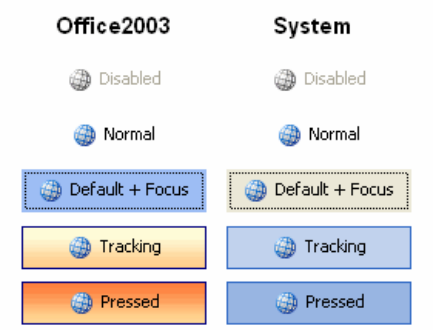


Figure 3 – ButtonStyle = LowProfile

KryptonCheckBox

The *KryptonCheckBox* control is great way to present the user with a choice such as *Yes/No* or *True/False*. This control combines check box functionality with the styling features of the *Krypton Toolkit*. The user can cycle between three states of checked, indeterminate and unchecked by pressing the check box. Place code in the *CheckStateChanged* event handler to perform actions based on the checked state. The content of the control is contained in the *Values* property. You can define *Text*, *ExtraText* and *Image* details within the *Values* property.

Appearance
The control uses a label style for presenting the text and image associated with the check box. The *LabelStyle* property has a default value of *NormalControl* giving the same appearance as a *KryptonLabel* instance. If you place the check box control onto a *Panel* background then you are recommended to change the *LabelStyle* to *NormalPanel* so that the label has an appropriate contrasting appearance.

You can use the *Orientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 1 for examples.

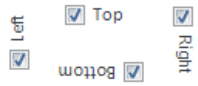


Figure 1 – Orientation Property

To alter the relative location of the check box use the *CheckPosition* property. The default is Left and shows the check box on the left side of the control values. Alternatively use the *Top*, *Bottom* or *Right* values as shown in Figure 2.

Figure 2 – CheckPosition Property

Checked & CheckState
To define the checked state of the control you can use either the *Checked* or the *CheckState* properties. The *Checked* property is a boolean property that returns *True* when the *CheckState* is *Checked*, otherwise it returns *False*. If you have defined the *ThreeState* property to be *False* then the check box can only ever be in the checked or unchecked states and so the *Checked* property is the most appropriate for setting or getting the current checked state. If however you have enabled the use of the indeterminate state by setting *ThreeState* to be *True* then you should use the *CheckState* property for getting and setting the current state, as it allows you to specify the additional indeterminate setting.

You can hook into the *CheckedChanged* and *CheckStateChanged* events in order to be notified when these two property values have been updated.

ThreeState
This defaults to *False* and so when you first create a new instance of the check box control it will toggle between checked and unchecked. Set this property to *True* if you also require the indeterminate state.

AutoCheck
Normally you will want the keyboard mnemonic shortcut and the mouse to automatically cycle around the check box states. If instead you would like to disable this automatic behavior then set this property to *False*, but then you will need to handle the state changes yourself.

Two States
Only two possible states of *Disabled* and *Normal* are used by the check box control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Note that only the content characteristics can be modified as the check box never has a border or background.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the text font in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Focus Override
If the control currently has the focus then the *OverrideFocus* settings are applied. This can occur when the control is in the *Normal* state. By default the override only alters the appearance so that a focus rectangle is drawn around the content so that the user can see that the control currently has the focus.

Examples of Appearance

Figure 3 shows the appearance when using the *Office 2007 - Blue* palette with the default settings. You can see each of the different *CheckState* values and the appearance when disabled, normal, tracking or pressed down.

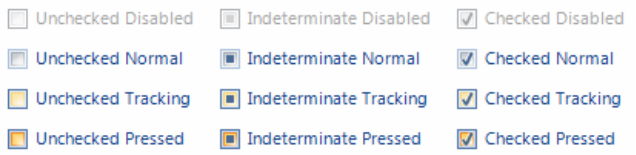


Figure 3 – CheckBox Appearance

KryptonCheckButton

Use a *KryptonCheckButton* control when you need to present the user with a binary choice such as *Yes/No* or *True/False*. This control combines the toggle button functionality with the styling features of the *Krypton Toolkit*. The user can toggle between checked and unchecked states by pressing the button. Place code in the *CheckedChanged* event handler to perform actions based on the checked state. The content of the control is contained in the *Values* property. You can define *Text*, *ExtraText* and *Image* details within the *Values* property.

Appearance
Use the *ButtonStyle* property to define the top level styling required for the appearance of the *KryptonCheckButton* control. The default value of *Standalone* gives a solid appearance that should be appropriate for use in most circumstances. Alternatively you can use the *LowProfile* setting for situations where you need a transparent border and background. See figures 2 and 3 below for examples of the visual difference. A third button style *ButtonSpec* is used by default for button specifications that are placed inside *KryptonHeader* and *KryptonHeaderGroup* controls. Buttons that appear on the caption area of a *KryptonForm* use the *Form* style. There are also custom styles that can be defined via a *KryptonPalette* for situations where you need to create variations on the styles already provided. Custom styles are named simply *Custom1*, *Custom2* and *Custom3*.

You can use the *Orientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 1 for examples.

Figure 1 – Orientation Property

Seven States
The button can be in one of seven possible states, *Disabled*, *Normal*, *CheckedNormal*, *Tracking*, *CheckedTracking*, *Pressed* and *CheckedPressed*. If the control has been disabled because the *Enabled* property is defined as *False* then the button will be in the *Disabled* state.

When enabled and unchecked the button will be in the *Normal* state until the user moves the mouse over the button at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the control then it enters the *Pressed* state. If the button is checked then the corresponding states would be *CheckedNormal*, *CheckedTracking* and *CheckedPressed*.

In order to customize the appearance of the control in each of the seven states you can use the properties *StateDisabled*, *StateNormal*, *StateCheckedNormal*, *StateTracking*, *StateCheckedTracking*, *StatePressed* and *StateCheckedPressed*. Each of these properties allows you to modify the background, border and content characteristics.

Common State
To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the border of the button to be 3 pixels wide with a rounding of 2 pixels and always red. Without the *StateCommon* property you would need to update the same three border settings in each of *StateDisabled*, *StateNormal*, *StateCheckedNormal*, *StateTracking*, *StateCheckedTracking*, *StatePressed* and *StateCheckedPressed* properties. Instead you can define the three border settings in just *StateCommon* and know they will be used whichever of the seven appearance states the button is using.

State Overrides
There are two additional properties called *OverrideDefault* and *OverrideFocus* that are involved in altering the appearance of the control. Notice that these start with the *Override* prefix instead of the usual *State*. This is because they do not relate to a specific control state such as *Normal* or *Tracking*. Instead they can be applied to any of the seven possible states and are used to override the appearance that would otherwise be shown.

Focus Override
If the control currently has the focus then the *OverrideFocus* settings are applied. This can occur when the control is in *Normal*, *CheckedNormal*, *Tracking*, *CheckedTracking*, *Pressed* or *CheckedPressed* states. By default the override only alters the appearance so that a focus rectangle is drawn around the content so that the user can see that the control currently has the focus.

Default Button Override

Each *Form* has the concept of a default button that is clicked when the user presses the *Enter* key. The *KryptonCheckButton* also supports the default button mechanism and *OverrideDefault* settings are applied to the appearance whenever the control has been designated as the default.

The following two images show the appearance of the *KryptonCheckButton* in the various different states for the two standard palettes of *Professional - Office 2003* and *Professional - System*. Each of the seven states *Disable*, *Normal*, *CheckedNormal*, *Tracking*, *CheckedTracking*, *Pressed* and *CheckedPressed*, are shown with an additional example to show the *OverrideDefault* and *OverrideFocus* applied to the Normal state.

AllowUncheck Property

This property has a default value of *True* and so when in the checked state the user is allowed to click the button to cause a transition to the unchecked state. If you want to prevent the user from unchecking the button then just set *AllowUncheck* to *False*. This is useful if you have a group of related check buttons and want to ensure that at least one is checked at all times. In that scenario you would use this property to prevent the user from unchecking the button and so prevent the outcome where none of the grouped buttons are checked.

Examples of Appearance

Figure 2 shows the appearance when the default *ButtonStyle* of *Standalone* is used. You can see that all the different states are drawn as solid controls. The third button down shows how the default button and focus overrides have affected the appearance of the *Normal* state above it.

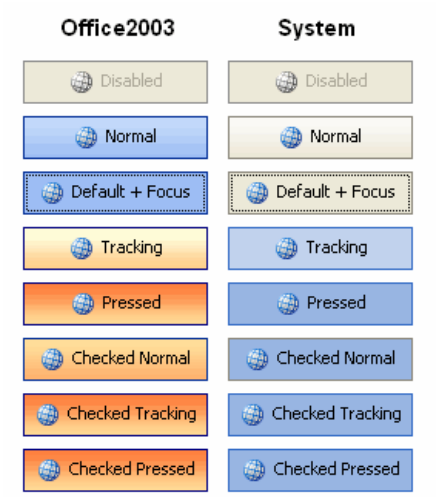


Figure 2 – ButtonStyle = Standalone

Figure 3 shows exactly the same states but this time the *ButtonStyle* is *LowProfile*. This time the background is transparent in the *Disabled* and *Normal* states. This button style is intended for use when you need to overlay a button onto a background and you want the button to be low profile and only light up when the user is interacting with it.

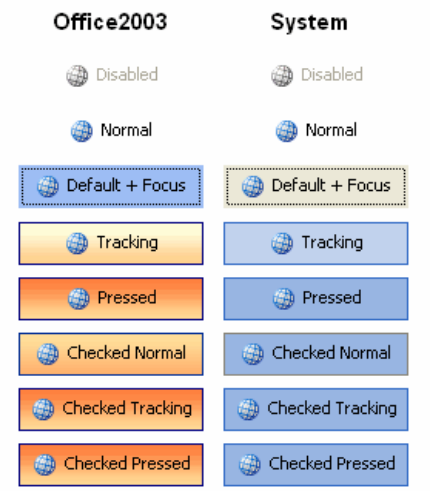


Figure 3 – ButtonStyle = LowProfile

KryptonCheckedListBox

Use the *KryptonCheckedListBox* when you need to present the user with a list of items that have associated check mark state. Use the *SelectionMode* property to determine how many items can be selected at a time. Set the *CheckOnClick* property if you require the checked state to toggle whenever the use clicks an item. Hook into the *SelectedIndexChanged* event to be notified when the selected item is changed.

Appearance

Use the *BackStyle*, *BorderStyle* and *ItemStyle* properties to alter the appearance of the control and the list items. The first two of these properties define the appearance of the overall control but the *ItemStyle* is used to define the display of the individual list items themselves. You can see in figure 1 the default appearance using the *Office 2007 - Blue* builtin palette.

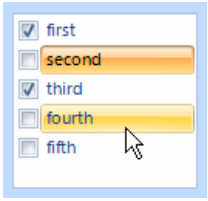


Figure 1 – Default Appearance

Eight States

There are eight different states relating to the control but not all the states are relevant to every part of the control.

The border and background of the control use just the *StateNormal*, *StateDisabled* or *StateActive* sets of properties. If the control has been disabled because the *Enabled* property is defined as *False* then the control always uses the *StateDisabled* values. When the control is active the *StateActive* properties are used and when not active the *StateNormal*. Being currently active means it has the focus or the mouse is currently over the control. Note that if the *AlwaysActive* property is defined as *True* then it ignores the *StateNormal* and always uses the *StateActive* regardless of the current active state of the control.

Drawing of the individual list items uses all the state properties except *StateActive*. This is because the list items are not drawn differently depending on the active state of the control. All the states that begin with the *StateChecked* name are used for items that are currently selected. When not selected an item uses one of the *StateNormal*, *StateTracking* or *StatePressed* properties.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the border of the button to be 3 pixels wide with a rounding of 2 pixels and always red. Without the StateCommon property you would need to update the same three border settings in each of *StateDisabled*, *StateNormal*, *StateTracking* etc properties. Instead you can define the three border settings in just *StateCommon* and know they will be used whichever of the states the checked list box happens to be using.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied to the item that has the focus indication. This can occur when the control is in *Normal*, *Tracking* or *Pressed* states. By default the override only alters the appearance so that a focus rectangle is drawn around the list item contents so that the user can see that the control currently has the focus.

AlwaysActive

This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The left instance does not have the mouse over it and the right instance does.

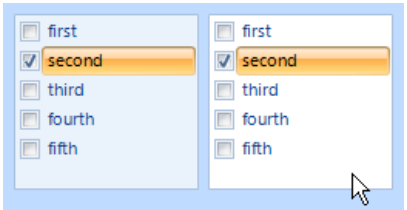


Figure 2 – InputControlStyle - Ribbon

KryptonListItem

You can add any object to the *Items* collection and it will use the *ToString()* method to recover the text for display. Alternatively you can add expose the *Krypton* interface *IContentValues* from your object and instead it will show the triple set of values that the interface exposes, two text strings and an image. To simplify the process you can create and add instances of the *KryptonListItem* class that exposes that interface for you. Figure 3 shows an example with several instances of this class added to the *Items* collection.

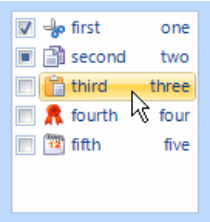


Figure 3 – Items collection containing KryptonListItem instances

KryptonColorButton

KryptonColorButton

This button allows the user to select a color value. It provides feedback by drawing the currently selected color as part of the image portion of the button contents and when pressed displays a context menu that allows the user to pick a new color value. It generates the *SelectedColorChanged* event whenever the user picks a new color. The *TrackColor* is fired when the context menu is being displayed and the user tracks over different color options, allowing the application to provide instant feedback during the color selection process. Hook into the *MoreColors* event if you need to provide your own custom dialog for selecting extra colors that are not displayed in the context menu.

The implementation actually provides all the functionality of the KryptonDropButton but with the extra events and properties needed to provide color selection capabilities. It is therefore recommended you read the documentation for the drop button. Note that the *KryptonContextMenu* property is not present on this control because it is managed by the color button and not provided by the developer. You can customize the contents of the displayed context menu by intercepting the *DropDown* event and then modifying the provided context menu before returning.

Appearance

Define the current color by using the *SelectedColor* property. This color will then be displayed in a rectangle that overlays the image portion of the button contents. The first three images of figure 1 show the display with *Red*, *Green* and *Blue* as the respective selected color. If the *Empty* value is selected then instead of drawing as a solid block only the border around the block is drawn in the *EmptyBorderColor* setting. This can be seen as the bottom image in figure 1. This makes sense because drawing the *Empty* color would result in nothing being drawn and so to ensure some feedback is presented to the user only the border of the rectangle is drawn. Change *EmptyBorderColor* if the default dark gray is not appropriate as the empty border.

By default the color block is drawn at the bottom of the image area. You can use the *SelectedRect* property to change this setting to something that matches the image you have decided to show. Figure 2 shows the default setting at the top following by two custom rectangles.

Important: If you don't provide an image for the content then you will note get the selected color displayed!

Figure 1 – SelectedColor and EmptyBorderColor Properties

Figure 2 – SelectedRect Property

Behavior

Figure 3 shows the context menu used with the default settings applied. This section describes how to use the various behavior properties to modify the contents and operation of the color button and the context menu. The five areas of the context menu, from top to bottom, are *Themes*, *Standard*, *Recent*, *NoColor* and *MoreColors*. Each of these areas can be turned off by using the associated visible property. So define *VisibleThemes* to be *False* in order to remove the display of the theme colors section at the top of the context menu. Remove the *No Color* option by setting *VisibleNoColor* to *False,* something you might want to do if your application needs to ensure a drawable color is always defined.

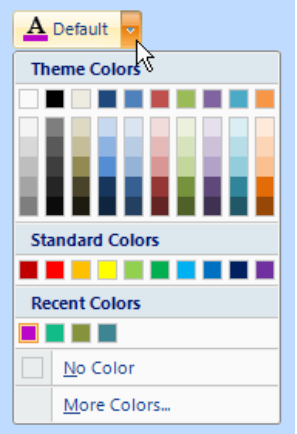


Figure 3 – Displayed Context Menu

In order to change the set of colors shown within the *Themes* and *Standard* areas you can modify the *SchemeThemes* and *SchemeStandard* properties, figure 4 shows these defined as *Basic16* and *Mono8* respectively. To alter the title strings above the color blocks you expand the *Strings* compound property at design time and then alter the appropriate properties as required.

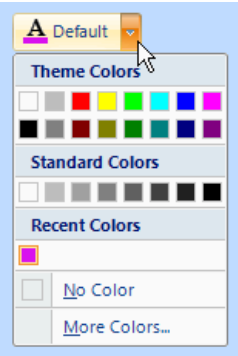


Figure 4 – Modified SchemeThemes and SchemeStandard

The purpose of the *Recent* section of the menu is two fold. First of all it provides a place for displaying the selected color if that color is not present in the two scheme areas above. This can happen if the user selects the *More Colors* option and uses it to find an unusual color. It also provides a memory of recent selections (that are not part of the schemes areas) so the user can pick a color that was recently used. Use the *MaxRecentColors* property to define the maximum number of color entries in this area. Any additional recent colors above this number are discarded. If you do not want the contents of the recent colors area automatically updated as selection changes occur then set the *AutoRecentColors* property to *False*.

Customizing the Context Menu

To alter the contents of the drop down context menu you need to hook into the *DropDown* event. This event provides a reference to the *KryptonContextMenu* instance that is about to be shown. Update this instance by adding your own context menu entries as required. Note that the next time this event is called the changes will still be present. You should remove the display of standard items by setting the appropriate visible property to false rather than removing it from the actual context menu instance.

Four States

The button can be in one of four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. If the control has been disabled because the *Enabled* property is defined as *False* then the button will be in the *Disabled* state.

When enabled the button will be in the *Normal* state until the user moves the mouse over the button at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the control then it enters the *Pressed* state. Note that if you have defined the drop down to be a *Splitter* then the two portions of the button can be in different states. For example, whilst the mouse is pressed over the drop arrow area that area will be shown in the *Pressed* state but the remainder will be drawn in the *Tracking* state.

In order to customize the appearance of the control in each of the four states you can use the properties *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed*. Each of these properties allows you to modify the background, border and content characteristics.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the border of the button to be 3 pixels wide with a rounding of 2 pixels and always red. Without the *StateCommon* property you would need to update the same three border settings in each of *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed* properties. Instead you can define the three border settings in just *StateCommon* and know they will be used whichever of the four appearance states the button happens to be using.

State Overrides

There are two additional properties called *OverrideDefault* and *OverrideFocus* that are involved in altering the appearance of the control. Notice that these start with the *Override* prefix instead of the usual *State*. This is because they do not relate to a specific control state such as *Normal* or *Tracking*. Instead they can be applied to any of the four possible states and are used to override the appearance that would otherwise be shown.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied. This can occur when the control is in *Normal*, *Tracking* or *Pressed* states. By default the override only alters the appearance so that a focus rectangle is drawn around the content so that the user can see that the control currently has the focus.

Default Button Override

Each *Form* has the concept of a default button that is clicked when the user presses the *Enter* key. The *KryptonButton* also supports the default button mechanism and *OverrideDefault* settings are applied to the appearance whenever the control has been designated as the default.

Examples of Appearance

Figure 5 shows the appearance of different states. The top example shows the color button without any mouse interaction. The second and third images show the mouse tracking over the two areas of the control and the last image shows the context menu displayed when the drop arrow is pressed.

Figure 5 – DropDown Appearance

KryptonComboBox

The *KryptonCheckBox* control is used to present display data in a drop-down combo box. The control appears in two parts, the top part is a text box that allows the user to type a list time. The second part is a pop up window displaying a list of options when the drop down button is pressed. This control uses the *Krypton* palette to obtain values for the drawing of the control.

Implementation
It is important to note that the implementation of the *Krypton* control is achieved by embedding a standard windows forms *ComboBox* control inside a custom control. As such the functionality of the combo box portion of the control is the same as that of the standard windows forms control. The *Krypton* implementation draws the border around the embedded control area and also allows for the display of *ButtonSpec* instances.

You can directly access the embedded control instance by using the *KryptonComboBox.ComboBox* property. This property is not exposed at design time but can be accessed directly using code. In most scenarios you should not need to access the underlying control as the majority of methods and properties for the embedded combo box are exposed directly by the *KryptonComboBox*. However if you need to implement drag and drop functionality you will need to hook directly into the events exposed by the embedded combo box and not the drag and drop events of the *Krypton* custom control.

Appearance
The *InputControlStyle* property has a default value of *Standalone* giving the same appearance as seen in figure 1. As the name suggests this is intended for use in a scenario where the control is used in a standalone fashion and used on something like a *KryptonPanel* or *KryptonGroup*. The *InputControlStyle* of *Ribbon* is intended for use when the control is present inside the *KryptonRibbon* and then needs a different appearance and operation.

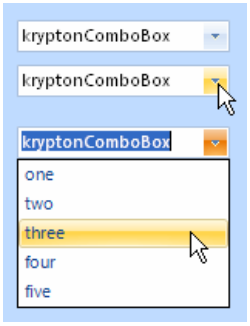


Figure 1 – InputControlStyle = Standalone

The *DropBackStyle* and *ItemStyle* are used to defined the background drawing and individual item drawing of the drop down list. Another property used to control the appearance is the *DropButtonStyle*. This has a default value of *InputControl* and is used to determine the appearance of the drop down button that is used to show the drop down list of the control. It is unlikely you would want to change this setting as the button appearance using other button styles would not look appropriate.

Three States
Only three possible states of *Disabled, Normal* or *Active* are used by the combo box control. In order to customize the appearance use the corresponding *StateDisabled, StateNormal* and *StateActive* properties. Each of these properties allows you to modify the background, border and content characteristics. Note that the control is restricted to the *Disabled* and *Active* states if the *AlwaysActive* property is defined as *True*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

AlwaysActive
This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The top instance does not have the mouse over it and the bottom instance does.

Figure 2 – InputControlStyle = Ribbon

ButtonSpecs
Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control. Figure 3 shows an example of a button specification that has been created to be positioned at the *Near* edge with a button style of *Standalone* and a button type of *Close*. You could then use this button to clear the contents of the control instead of requiring the user to manually clear the contents. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

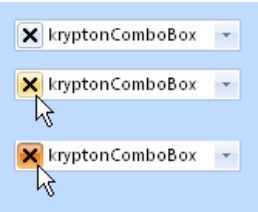


Figure 3 – InputControlStyle = Ribbon

AllowButtonSpecTooltips
By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

KryptonDataGridView

The *KryptonDataGridView* control derives from the standard *DataGridView* that comes with .NET 2.0. This allows you to take advantage of all the features present in the *DataGridView* control but with the advantage of drawing in the *Krypton* palette style. As with all *Krypton* controls you can customize the appearance by providing a custom palette or by modifying the individual state appearance properties. For information about non-appearance aspects of the control you should refer to the *DataGridView* control documentation.

Grid Styles

Select the *KryptonDataGridView* control at design time and using the properties window you can navigate to the *GridStyles* compound property. This property is used to specify the *Krypton* styles used for drawing the different elements of the grid. There are three built-in styles called *List*, *Sheet* and *Custom1*. Usually you would modify all the grid elements to use the same *List*, *Sheet* or *Custom1* setting so that the appearance is consistent.

For example, if you look at the left image in *Figure 1* you can see the default settings where each grid element is using the *List* setting. Changing just the *Style* property will cause all the other properties to change to the correct matching style. The right image of *Figure 1* shows where *Style* has been updated to *Sheet*. So if you would like all the properties to be in sync then use the *Style* property.

However, if you would like to mix and match the styles of the different grid elements then you can modify them individually. For example you might change *StyleDataCells* to be *Sheet* and then alter the *StyleRow* to be *List*. In that scenario the *Style* property will automatically update itself to become *Mixed* as the settings of the grid element properties are a mixture of styles and not consistent.

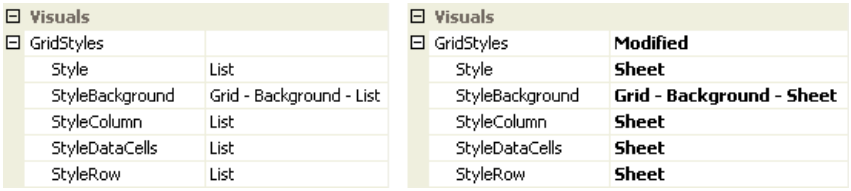


Figure 1 - Style = List & Style - Sheet

You can see the visual different between the *List* and *Sheet* styles by looking at *Figure 2*. The left image is the *List* style when the *Office 2007 - Blue* palette is used and the right image is the same palette but with the *Sheet* style.

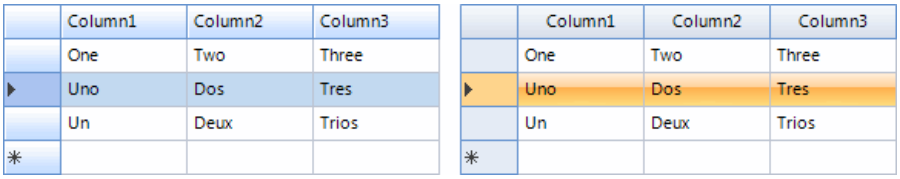


Figure 2 - Grid Appearance for List & Sheet styles

HideOuterBorders

Unlike the standard *DataGridView* control the *Krypton* version never draws a border around the entire control area. This is because the *DataGridView* control does not provide an effective method of overriding the drawing of the control border and so it has been always removed. Many developers may find that they do need a *Krypton* appropriate border around the control. To achieve the border you can simply place the *KryptonDataGridView* instance inside a *KryptonGroup* instance and modify the grid to fill the entire group area by setting the *Dock* property to *Fill*.

This solution does not always give an appropriate appearance. You can see in the left image of *Figure 3* that the border around the entire control clashes with the border around the grid cells. By setting the *HideOuterBorders* to True you prevent the drawing of the cells borders where they are adjacent to the group border. The right image in *Figure 3* shows the resulting more aesthetic result.

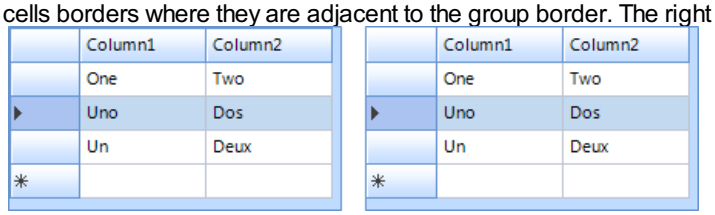


Figure 3 - HideOuterBorders = False & True

Five States

There are 5 different states that grid elements can be in at any given time. These are *Normal*, *Disabled*, *Tracking*, *Pressed* and *Selected*. Note however that not all the different elements use all the possible states. All the grid elements of *DataCells*, *HeaderColumn*, *HeaderRow* and *Background* make use of the *Normal* and *Disabled* states. All except the *Background* make use of the *Selected* state. Only the *HeaderColumn* and *HeaderRow* use the *Tracking* and *Pressed* states. To find the appearance

properties for a given state just look in the *Visuals* category of the properties window and expand the relevant named compound property such as *StateNormal*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define a text color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Overriding Cell Styles

It is common when using a grid control to override the appearance of individual cells or whole columns of cells. For example you might have a column that contains currency values and want to show the negative values in red and the positive values in black. You can do this using the *KryptonDataGridView* control in the same way that you would for the standard *DataGridView* control.

To modify the *Font*, *BackColor*, *ForeColor*, *SelectionBackColor* and *SelectionForeColor* for a column you would use the properties window to find and edit the *Columns* collection. After selecting the column instance of interest you would alter the *DefaultCellStyle* from the editing dialog and then alter the properties as required. This is exactly the same process as for the standard *DataGridView* control.

Updating the cell style for an individual cell is done via code. You just access the cell style associated with the individual cell and change the property of interest. For example you could use the following code to change the text color to red for the first cell in a grid instance.

```
kryptonDataGridView1.Rows[0].Cells[0].Style.ForeColor = Colors.Red;
```

Overriding the cell styles works because the *KryptonDataGridView* only uses the palette defined colors when the cell style for a cell has been explicitly specified.

Defining Grid Columns

At design time you can specify the number and type of columns shown by the grid. You should always us the Krypton specified version of the column type whenever possible. For example, instead of using the *DataGridViewTextBoxColumn* instead select the *KryptonDataGridViewTextBoxColumn*. This ensures the column cells are drawn in a manner consistent with the Krypton palette. The nine possible Krypton specific types are:-

- *KryptonDataGridViewTextBoxColumn*
- *KryptonDataGridViewMaskedTextBoxColumn*
- *KryptonDataGridViewComboBoxColumn*
- *KryptonDataGridViewNumericUpDownColumn*
- *KryptonDataGridViewDomainUpDownColumn*
- *KryptonDataGridViewDateTimePickerColumn*
- *KryptonDataGridViewCheckBoxColumn*
- *KryptonDataGridViewButtonColumn*
- *KryptonDataGridViewLinkColumn*

When adding a new column definition at design time you will be presented with a ComboBox that lists all available column types. Figure 4 shows an example of this scenario and shows where the *Krypton* set of columns are displayed within the available options.

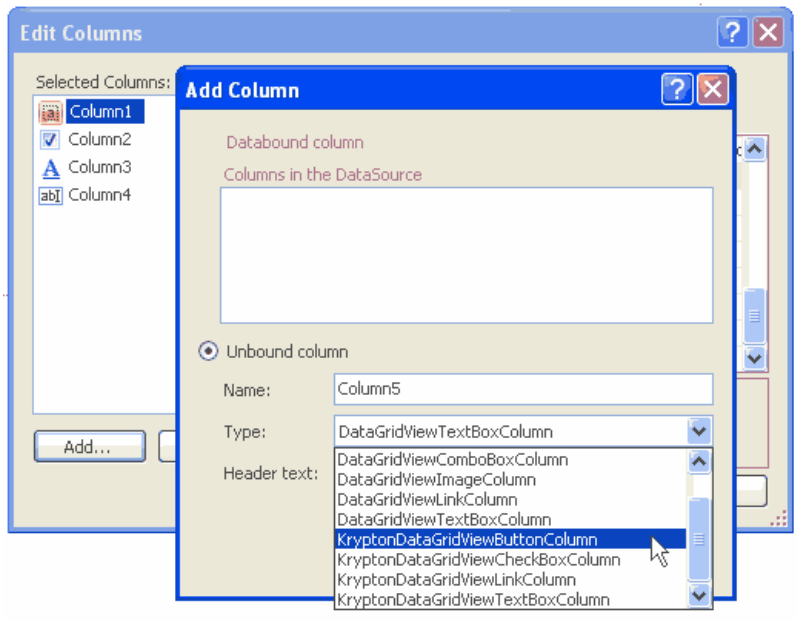


Figure 4 - Krypton Specific Columns

KryptonDateTimePicker
KryptonDateTimePicker

The *KryptonDateTimePicker* control allows the user to select a single item from a list of dates or times. When used to represent a date, it appears in two parts: a drop-down list with a date represented in text, and a grid appears when you click on the down-arrow next to the list. This control uses the Krypton palette to obtain values for the drawing of the control.

Appearance

The *InputControlStyle* property has a default value of *Standalone* giving the same appearance as seen in figure 1. As the name suggests this is intended for use in a scenario where the control is used in a standalone fashion and used on something like a *KryptonPanel* or *KryptonGroup*. The *InputControlStyle* of *Ribbon* is intended for use when the control is present inside the *KryptonRibbon* and then needs a different appearance and operation.

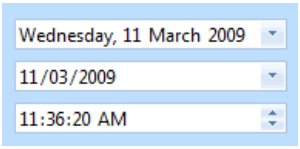


Figure 1 – InputControlStyle = Standalone

Three States

Only three possible states of *Disabled, Normal* or *Active* are used by the date time picker control. In order to customize the appearance use the corresponding *StateDisabled, StateNormal* and *StateActive* properties. Each of these properties allows you to modify the background, border and content characteristics. Note that the control is restricted to the *Disabled* and *Active* states if the *AlwaysActive* property is defined as *True*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

AlwaysActive

This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The top instance does not have the mouse over it and the bottom instance does.

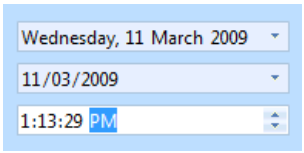


Figure 2 – InputControlStyle = Ribbon

ButtonSpecs

Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control. Figure 3 shows an example of a button specification that has been created to be positioned at the *Far* edge with a button style of *ButtonSpec* and a button type of *Context*. You could then use this button to show a context menu with additional options relevant to the entry field. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

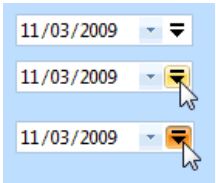


Figure 3 – KryptonDateTimePicker with ButtonSpec definition

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

KryptonDomainUpDown

The *KryptonDomainUpDown* control allows you to display text values with the ability to use spin buttons to change the value up and down a predefined list. This control uses the *Krypton* palette to obtain values for the drawing of the control.

Implementation
It is important to note that the implementation of the *Krypton* control is achieved by embedding a standard windows forms *DomainUpDown* control inside a custom control. As such the functionality of the edit portion of the control is the same as that of the standard windows forms control. The *Krypton* implementation draws the border around the embedded control area and also allows for the display of *ButtonSpec* instances.

You can directly access the embedded control instance by using the *KryptonDomainUpDown.DomainUpDown* property. This property is not exposed at design time but can be accessed directly using code. In most scenarios you should not need to access the underlying control as the majority of methods and properties for the embedded domain control are exposed directly by the *KryptonDomainUpDown*. However if you need to implement drag and drop functionality you will need to hook directly into the events exposed by the embedded control and not the drag and drop events of the *Krypton* custom control.

Appearance
The *InputControlStyle* property has a default value of *Standalone* giving the same appearance as seen in figure 1. As the name suggests this is intended for use in a scenario where the control is used in a standalone fashion and used on something like a *KryptonPanel* or *KryptonGroup*. The *InputControlStyle* of *Ribbon* is intended for use when the control is present inside the *KryptonRibbon* and then needs a different appearance and operation.

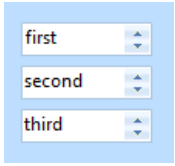


Figure 1 – InputControlStyle = Standalone

Three States
Only three possible states of *Disabled, Normal* or *Active* are used by the domain control. In order to customize the appearance use the corresponding *StateDisabled, StateNormal* and *StateActive* properties. Each of these properties allows you to modify the background, border and content characteristics. Note that the control is restricted to the *Disabled* and *Active* states if the *AlwaysActive* property is defined as *True*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

AlwaysActive
This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The top instance does not have the mouse over it and the bottom instance does.

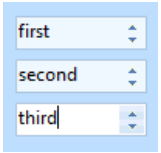


Figure 2 – InputControlStyle = Ribbon

ButtonSpecs
Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control. Figure 3 shows an example of a button specification that has been created to be positioned at the *Far* edge with a button style of *ButtonSpec* and a button type of *Context*. You could then use this button to show a context menu with additional options relevant to the entry field. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

Figure 3 – KryptonDomainUpDown with ButtonSpec definition

AllowButtonSpecTooltips
By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

KryptonDropDown
KryptonDropButton

This specialized button control is used to display a context menu when the button is pressed. An arrow symbol is shown inside the control as a visual indication of the drop down capability. If you choose to set the *Splitter* property then the button is split into two sections, a small area that contains the drop down arrow and the remained that acts like a standard button. Pressing the button causes the *Click* event to be generated and clicking the drop down portion of the control fires the *DropDown* event. The content of the control is contained in the *Values* property. You can define *Text*, *ExtraText* and *Image* details within the *Values* property.

Appearance

Use the *Splitter* property to define if the drop down button should be placed in its own separate area. Figure 1 shows the visual difference between the two possible settings. You can use the *ButtonOrientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 2 for examples. To alter the position of the drop down arrow use the *DropDownPosition* property as seen in figure 3. If you need to change the direction the arrow points, and therefore the location the context menu appears, then update *DropDownPosition* as shown in figure 4 below.

Update *ButtonStyle* to define the top level styling required for the appearance of the *KryptonDropButton* control. The default value of *Standalone* gives a solid appearance that should be appropriate for use in most circumstances. Alternatively you can use the *LowProfile* setting for situations where you need a transparent border and background. There are also custom styles that can be defined via a *KryptonPalette* for situations where you need to create variations on the styles already provided. Custom styles are named simply *Custom1*, *Custom2* and *Custom3*.

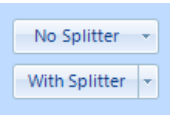


Figure 1 – Splitter Property

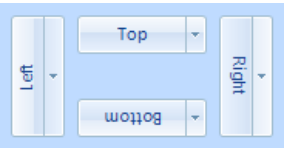


Figure 2 – ButtonOrientation Property

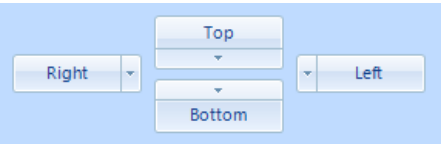


Figure 3 – DropDownPosition Property

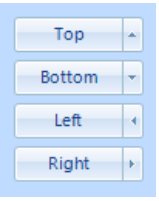


Figure 4 – DropDownOrientation Property

Four States

The button can be in one of four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. If the control has been disabled because the *Enabled* property is defined as *False* then the button will be in the *Disabled* state.

When enabled the button will be in the *Normal* state until the user moves the mouse over the button at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the control then it enters the *Pressed* state. Note that if you have defined the drop down to be a *Splitter* then the two portions of the button can be in different states. For example, whilst the mouse is pressed over the drop arrow area that area will be shown in the *Pressed* state but the remainder will be drawn in the *Tracking* state.

In order to customize the appearance of the control in each of the four states you can use the properties *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed*. Each of these properties allows you to modify the background, border and content characteristics.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the border of the button to be 3 pixels wide with a rounding of 2 pixels and always red. Without the *StateCommon* property you would need to update the same three border settings in each of *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed* properties. Instead you can define the three border settings in just *StateCommon* and know they will be used whichever of the four appearance states the button happens to be using.

State Overrides

There are two additional properties called *OverrideDefault* and *OverrideFocus* that are involved in altering the appearance of the control. Notice that these start with the *Override* prefix instead of the usual *State*. This is because they do not relate to a specific control state such as *Normal* or *Tracking*. Instead they can be applied to any of the four possible states and are used to override the appearance that would otherwise be shown.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied. This can occur when the control is in *Normal*, *Tracking* or *Pressed* states. By default the override only alters the appearance so that a focus rectangle is drawn around the content so that the user can see that the control currently has the focus.

Default Button Override

Each *Form* has the concept of a default button that is clicked when the user presses the *Enter* key. The *KryptonButton* also supports the default button mechanism and *OverrideDefault* settings are applied to the appearance whenever the control has been designated as the default.

Examples of Appearance

Figure 5 shows the appearance of different states. The top example shows the drop button without any mouse interaction. The second and third images show the mouse tracking over the two areas of the control and the last image shows the context menu displayed when the drop arrow is pressed.

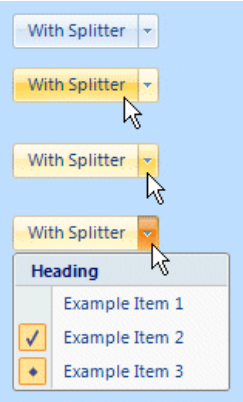


Figure 5 – DropDown Appearance

KryptonGroup

Use the *KryptonGroup* control when you need to group related controls together. For example, you can use groups to subdivide a form into distinct areas. Moving the group will cause all the contained controls to also be moved along with it as it acts as a container. This control is similar to the *KryptonPanel* except it provides a border as well as a background.

The *KryptonPanel* is more suited towards providing the background for large sections of the client area. *KryptonGroup* is more suitable for grouping a small number of related controls together.

Appearance

The *GroupBackStyle* and *GroupBorderStyle* properties are used to define the top level styling required for the appearance of the *KryptonGroup* control. The default value of *ControlClient* for both properties gives an appearance appropriate for grouping together related controls. Alternatively use the *ControlAlternate* setting for a group that needs to stand out. There is also a custom style that can be defined via a *KryptonPalette* for situations where you need to create a variation on the styles already provided. The custom style is called simply *Custom1*.

Two States

Only two possible states of *Disabled* and *Normal* are used by the group control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Note that only the background and border characteristics can be modified as the group control never has a content value.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the background color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Examples of Appearance

Figure 1 shows the appearance when *GroupBackStyle* and *GroupBorderStyle* are both defined as the default *Control1*. You can see that the *Disabled* and *Normal* states have the same appearance. If you need to show the group is disabled then you can alter the *StateDisabled* property as required.

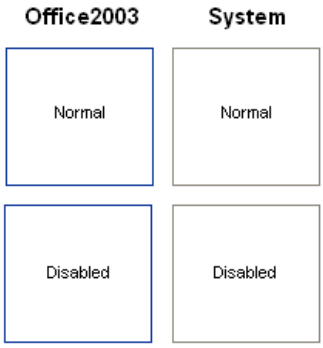


Figure 1 – KryptonGroup examples

KryptonGroupBox

Use the *KryptonGroupBox* control when you need to group related controls together and provide a caption. For example, you can use group boxes to subdivide a form into distinct areas. Moving the group will cause all the contained controls to also be moved along with it as it acts as a container. This control is similar to the *KryptonGroup* except it provides a caption as well as a grouping ability.

Appearance

The *GroupBackStyle* and *GroupBorderStyle* properties are used to define the level styling required for the background and border areas of the *KryptonGroupBox* control. The default value of *ControlGroupBox* for both properties gives an appearance appropriate for grouping together related controls. The style of the caption is defined by the *CaptionStyle* property and this default to *CaptionPanel*. By default the *CaptionOverlap* property is defined as *50%* and so causes the group border to be drawn halfway down the caption. If you would prefer the caption to be entirely outside of the border then set this property to *100%*. To place the caption entirely inside the border you can set the *CaptionOverlap* to *0%*.

Two States

Only two possible states of *Disabled* and *Normal* are used by the group box control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the background color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Examples of Appearance

Figure 1 shows the appearance when *GroupBackStyle* and *GroupBorderStyle* are both defined as the default *ControlGroupBox*.

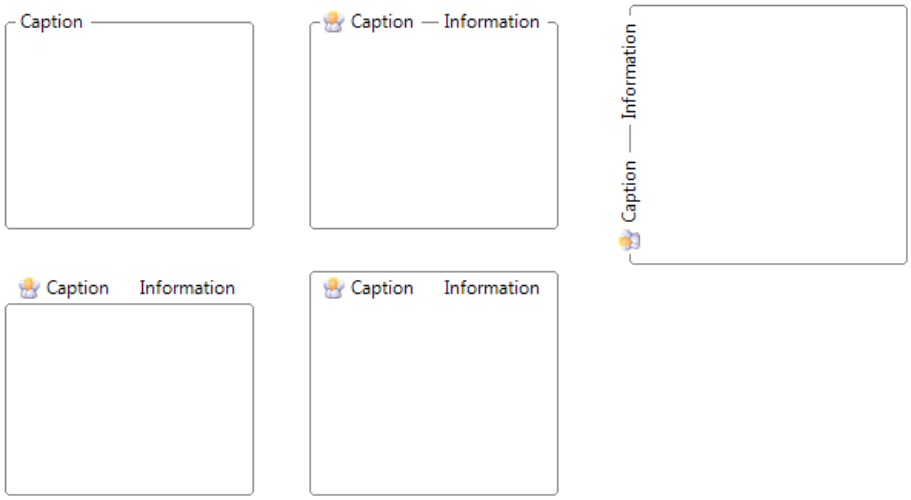


Figure 1 – KryptonGroupBox examples

KryptonHeader

Use the *KryptonHeader* control to create a heading that provides a description for a section of your form. For example, you can place a header at the top of an area of related controls and provide a name and description that indicates the purpose of the related controls. The content of the control is contained in the *Values* property. You can define *Heading*, *Description* and *Image* details within the *Values* property.

Appearance

The *HeaderStyle* property is used to define the top level styling required for the appearance of the *KryptonHeader* control. The default value of *Primary* gives a bold header style that should be appropriate for use as a section header. Alternatively use the *Secondary* setting for less prominent text. See figures 2 and 3 for examples of the visual difference. The *Form* style is used for the caption area by the *KryptonForm* control. There are also custom styles that can be defined via a *KryptonPalette* for situations where you need to create variations on the styles already provided. Custom styles are named simply *Custom1 and Custom2*.
You can use the *Orientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 1 for examples.

Figure 1 – Orientation Property

Two States

Only two possible states of *Disabled* and *Normal* are used by the header control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Each of these properties allows you to modify the background, border and content characteristics.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the text font in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

ButtonSpecs

You can add buttons to the header by modifying the *ButtonSpecs* collection exposed by the *KryptonHeader*. Each *ButtonSpec* entry in the collection describes a single button for display on the header. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances. See the ButtonSpec section for more details. Figure 2 shows an example of a *KryptonHeader* with buttons.

Figure 2 – KryptonHeader with ButtonSpecs

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

UseMnemonic

When this property is defined as *True* for the *KryptonHeader* it will check each of the *ButtonSpec* instances in the *ButtonSpecs* collection. If the *Text* or *ExtraText* in a *ButtonSpec* matches the incoming mnemonic then the *Click* event for the *ButtonSpec* will be fired.

Examples of Appearance

Figure 3 shows the *KrytonHeader* in the *Normal* and *Disabled* states for the *HeaderStyle* of *Primary.* The left column shows examples using the *Professional - Office 2003* palette mode and the right column the *Professional - System* palette mode.
Figure 4 shows the same except where the *HeaderStyle* has been changed to Secondary.

Figure 3 – HeaderStyle = Primary

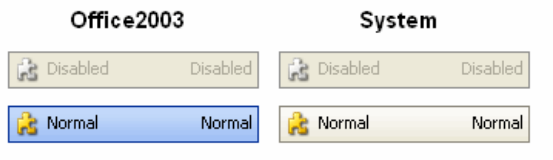


Figure 4 – HeaderStyle = Secondary

KryptonHeaderGroup

Use the *KryptonHeaderGroup* control to combine the benefits of the *KryptonHeader* and the *KryptonGroup* into one. When you need to group together related controls and provide a heading for them then the *KryptonHeaderGroup* should be used. You can position and define two headers within the header group control.

The content of the control is contained in the *ValuesPrimary* and *ValuesSecondary* properties. You can define *Heading*, *Description* and *Image* details within each of the *ValuesPrimary* and *ValuesSecondary* properties.

Appearance

The *GroupBackStyle* and *GroupBorderStyle* properties are used to determine the styling of the background and border of the main control. The group border is the border that encloses the entire control and does not control the border drawing of the individual headers. The group background is the client area that is used to contain child controls and does not control the background drawing of the individual headers.

To control the styling of the first header use the *HeaderStylePrimary* property and *HeaderStyleSecondary* for the second header. These have default values of *Primary* and *Secondary* respectively.

By default the primary header is positioned at the top of the control and the secondary header at the bottom. You can change this by altering the *HeaderPositionPrimary* and *HeaderPositionSecondary* values as can be seen in the examples in figure 1.

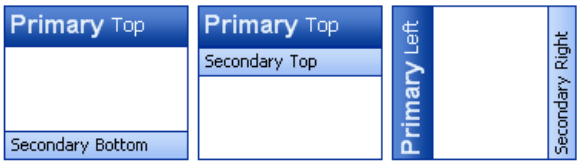


Figure 1 – HeaderPositionPrimary and HeaderPositionSecondary

If you do not require both headers to be displayed then just modify the *HeaderVisiblePrimary* and *HeaderVisibleSecondary* properties as appropriate. By combining the style, position and visibility properties you should be able to customize the layout and appearance to whatever you need.

Collapsing and Expanding Groups

By default the *KryptonHeaderGroup* has a *Collapsed* property value of *False,* indicating that the client area of the group should be shown. If you set *Collapsed* to *True* then the client area will be hidden from view. It will also use the *CollapsedTarget* property to decide on which headers should be visible in the collapse state. The default value is *CollapseToPrimaryHeader* and so will cause the primary header to be shown and the secondary header to be hidden. You can use the alternative values of *CollapseToSecondaryHeader* or *CollapseToBothHeaders* if you need an alternative collapsed appearance.

Note that the size of the header group will not change when you toggle the *Collapsed* property unless you set the *AutoSize* property to be *True*. The most common way of implementing an auto sizing header group is to dock it against one of the form edges. You will also want to modify the *MinimumSize* and *MaximumSize* values for the panel contained inside the header group (accessed from the *Panel* property of the *KryptonHeaderGroup* control). This is to ensure that when *AutoSize* is used it will calculate the required size appropriately.

The last property involved in collapsing functionality is called *AuotCollapseArrow*and defaults to *True*. If your *KryptonHeaderGroup* control has a *ButtonSpec* defined that has a *Type* of *ArrowUp*, *ArrowDown*, *ArrowLeft* or *ArrowRight* the it will automatically toggle the collapsed state when that *ButtonSpec* is clicked. It also inverts the *Type* and so *ArrowUp* would be automatically changed into *ArrowDown*. This property means you can implement a fully working collapsing/expanding header group without needing to write a single line of code. Check out the Expanding HeaderGroups (DockStyle) tutorial to see the detailed steps.

Two States

Only two possible states of *Disabled* and *Normal* are used by the header control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Each of these properties allows you to modify the background and border of the control as well as the background, border and content of the headers.

An additional property called *OverlayHeaders* is available in each of the state properties and determines if the headers are draw over the top of the group. Figure 2 shows an example of both settings where the border of the control is red and two pixels thick. In the first image you can see that *OverlayHeaders* is *True* and so the headers are drawn over the red border. In the second image the property is *False* and now the red border is drawn on top of the headers.

Figure 2 – OverlayHeaders True and False

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the text font in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

ButtonSpecs

You can add buttons to the individual headers by modifying the *ButtonSpecs* collection exposed by the *KryptonHeaderGroup*. Each *ButtonSpec* entry in the collection describes a single button for display on one of the headers. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances. See the ButtonSpec section for more details. Figure 2 shows an example of a *KryptonHeaderGroup* with buttons.

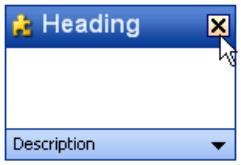


Figure 2 – KryptonHeaderGroup with ButtonSpecs

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

UseMnemonic

When this property is defined as *True* for the *KryptonHeaderGroup* it will check each of the *ButtonSpec* instances in the *ButtonSpecs* collection. If the *Text* or *ExtraText* in a *ButtonSpec* matches the incoming mnemonic then the *Click* event for the *ButtonSpec* will be fired.

Examples of Appearance

Figure 3 shows the header group in the *Disabled* and *Normal* states for the *Professional - Office 2003* and *Professional - System* palette modes.

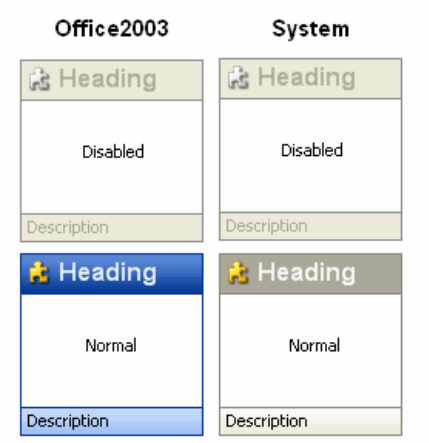


Figure 3 – Examples of appearance

KryptonLabel
KryptonLabel

Use the *KryptonLabel* control when you need display text and images combined with the styling features of the *Krypton Toolkit*. For example, you can use the label to add descriptive captions to other controls such as text boxes or list boxes. The content of the control is contained in the *Values* property. You can define *Text*, *ExtraText* and *Image* details within the *Values* property.

Appearance
The *LabelStyle* property is used to define the top level styling required for the appearance of the *KryptonLabel* control. The four standard values are *NormalControl*, *TitleControl*, *NormalPanel* and *TitlePanel*. It is important to understand when to use which style in order to get the correct appearance when switching to different palettes.
When using a label that is positioned with on a *Control* style background, such as *ControlClient* or *ControlAlternate*, then you should use the *NormalControl* or *TitleControl* styles. A good example would be placing label instances in the client area of a *KryptonHeaderGroup,* as the header group has a default background style in the client area of *ControlClient*. If however you are placing label instances onto a *KryptonPanel* then you should use the *NormalPanel* or *TitlePanel* styles. It is easy to forget to set the appropriate style because most of the builtin palettes have colors that look fine on either a *Control* or *Panel* style background. But if you use the *Office 2007 - Black* palette then it will fail to appear correctly as the colors are radically different. In that scenario a *NormalControl* on a *Panel* background would be invisible!

The *NormalControl* and *NormalPanel* styles give a standard text appearance appropriate for use as a caption for other controls. Alternatively use the *TitleControl/TitlePanel* settings for use as a section header where the text needs to be more prominent. See figure 2 for examples of the visual difference. There are also custom styles that can be defined via a *KryptonPalette* for situations where you need to create variations on the styles already provided. Custom styles are named simply *Custom1*, *Custom2* and *Custom3*.

You can use the *Orientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 1 for examples.

Figure 1 – Orientation Property

Two States
Only two possible states of *Disabled* and *Normal* are used by the label control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Note that only the content characteristics can be modified as the label control never has a border or background.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the text font in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

UseMnemonic & Target
If you provide an ampersand in the label text and also set the *UseMnemonic* property to *True* then the label text will show the character after the ampersand as underlined. This indicates to the user that pressing the mnemonic character will cause an action to occur. As the label control cannot interact with the user when the character is pressed an additional *Target* property is used to indicate the control you would like to take the focus instead.

This is useful when you have a label control used as a description of an input field. For example, you might have a label with the text defined as '&Name' positioned next to a text box control. You would set the *Target* property to be a reference to the text box instance. Now when the user presses the mnemonic character N it would set focus to the text box.

Examples of Appearance
Figure 2 shows the appearance of the label for both *LabelStyle* values of *Title* and *Normal*. In each case the label is first shown in the *Disabled* state and then the *Normal* state.

Figure 2 – Example appearance

KryptonLinkLabel

KryptonLinkLabel

Use the *KryptonLinkLabel* control when you need to display a hyper link in conjunction with the styling features of the Krypton Toolkit. This control allows the user to initiate an action by pressing the hyper link in order to generate a LinkClicked event. The content of the control is contained in the *Values* property. You can define *Text*, *ExtraText* and *Image* details within the *Values* property.

Appearance

The *LabelStyle* property is used to define the top level styling required for the appearance of the *KryptonLinkLabel* control. The four standard values are *NormalControl*, *TitleControl*, *NormalPanel* and *TitlePanel*. It is important to understand when to use which style in order to get the correct appearance when switching to different palettes.

When using a label that is positioned with on a *Control* style background, such as *ControlClient* or *ControlAlternate*, then you should use the *NormalControl* or *TitleControl* styles. A good example would be placing label instances in the client area of a *KryptonHeaderGroup,* as the header group has a default background style in the client area of *ControlClient*. If however you are placing label instances onto a *KryptonPanel* then you should use the *NormalPanel* or *TitlePanel* styles. It is easy to forget to set the appropriate style because most of the builtin palettes have colors that look fine on either a *Control* or *Panel* style background. But if you use the *Office 2007 - Black* palette then it will fail to appear correctly as the colors are radically different. In that scenario a *NormalControl* on a *Panel* background would be invisible!

The *NormalControl* and *NormalPanel* styles give a standard text appearance appropriate for use as a caption for other controls. Alternatively use the *TitleControl/TitlePanel* settings for use as a section header where the text needs to be more prominent. See figure 2 for examples of the visual difference. There are also custom styles that can be defined via a *KryptonPalette* for situations where you need to create variations on the styles already provided. Custom styles are named simply *Custom1*, *Custom2* and *Custom3*.

You can use the *Orientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 1 for examples.

Figure 1 – Orientation Property

Two States

Only two possible states of *Disabled* and *Normal* are used by the label control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Note that only the content characteristics can be modified as the label control never has a border or background.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the text font in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

State Overrides

There are four additional properties called *OverrideFocus*, *OverrideNotVisited*, *OverrideVisited* and *OverridePressed* that are involved in altering the appearance of the control. Notice that these start with the *Override* prefix instead of the usual *State*. This is because they do not relate to a specific control state such as *Normal* or *Pressed*. Instead they can be applied to any of the possible states and are used to override the appearance that would otherwise be shown.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied. By default the override only alters the appearance so that a focus rectangle is drawn around the content so that the user can see that the control currently has the focus.

NotVisited & Visited Overrides

The *OverrideNotVisited* override is used to alter the *StateNormal* settings when the hyper link has not yet been visited. The *LinkVisited* property on the control is used to specify when the link has been visited or not. When *LinkVisited* is *False* this override is applied. When *LinkVisited* is *True* the *OverrideVisited* is applied instead.

Pressed Override

To customize the appearance when the user is using the mouse to press down the hyper link use the *OverridePressed* settings. Note that the *OverridePressed* can be applied at the same time as any of the other overrides.

Link Behavior

There are three options for deciding if the hyper link should be underlined. The default value of the *LinkBehavior* property is *AlwaysUnderline* and so the hyper link has an underline at all times. If you prefer to have an underline only when the mouse is hovering over the hyper link then change the property to *HoverUnderline*. Finally the *NeverUnderline* is appropriate when you never want the text underlined.

Examples of Appearance

Figure 2 shows the appearance of the link label for the *LabelStyle* value of *NormalControl*.

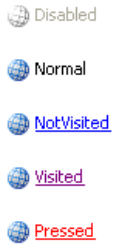


Figure 2 – LabelStyle = Normal

KryptonListBox

Use the *KryptonListBox* when you need to present the user with a list from which the user can select one or more items. If the number of items exceeds the number that can be displayed then a scroll bar is automatically shown. Use the *SelectionMode* property to determine how many items can be selected at a time. Hook into the *SelectedIndexChanged* event to be notified when the selected item is changed.

Appearance

Use the *BackStyle*, *BorderStyle* and *ItemStyle* properties to alter the appearance of the control and the list items. The first two of these properties define the appearance of the overall control but the *ItemStyle* is used to define the display of the individual list items themselves. You can see in figure 1 the default appearance using the *Office 2007 - Blue* builtin palette.

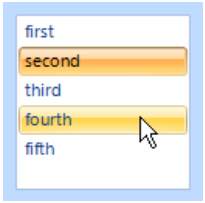


Figure 1 – Default Appearance

Eight States

There are eight different states relating to the control but not all the states are relevant to every part of the control.

The border and background of the control use just the *StateNormal*, *StateDisabled* or *StateActive* sets of properties. If the control has been disabled because the *Enabled* property is defined as *False* then the control always uses the *StateDisabled* values. When the control is active the *StateActive* properties are used and when not active the *StateNormal*. Being currently active means it has the focus or the mouse is currently over the control. Note that if the *AlwaysActive* property is defined as *True* then it ignores the *StateNormal* and always uses the *StateActive* regardless of the current active state of the control.

Drawing of the individual list items uses all the state properties except *StateActive*. This is because the list items are not drawn differently depending on the active state of the control. All the states that begin with the *StateChecked* name are used for items that are currently selected. When not selected an item uses one of the *StateNormal*, *StateTracking* or *StatePressed* properties.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the border of the button to be 3 pixels wide with a rounding of 2 pixels and always red. Without the StateCommon property you would need to update the same three border settings in each of *StateDisabled*, *StateNormal*, *StateTracking* etc properties. Instead you can define the three border settings in just *StateCommon* and know they will be used whichever of the states the list box happens to be using.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied to the item that has the focus indication. This can occur when the control is in *Normal*, *Tracking* or *Pressed* states. By default the override only alters the appearance so that a focus rectangle is drawn around the list item contents so that the user can see that the control currently has the focus.

AlwaysActive

This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The left instance does not have the mouse over it and the right instance does.

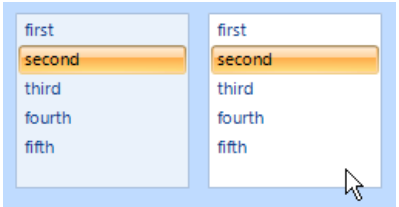


Figure 2 – InputControlStyle - Ribbon

KryptonListItem

You can add any object to the *Items* collection and it will use the *ToString()* method to recover the text for display. Alternatively you can add expose the *Krypton* interface *IContentValues* from your object and instead it will show the triple set of values that the interface exposes, two text strings and an image. To simplify the process you can create and add instances of the *KryptonListItem* class that exposes that interface for you. Figure 3 shows an example with several instances of this class added to the *Items* collection.

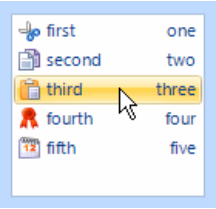


Figure 3 – Items collection containing KryptonListItem instances

KryptonMaskedTextBox

The *KryptonMaskedTextBox* control to distinguish between proper and improper user input. It allows the specification of a mask that describes the pattern of input that is allowed to be entered. This control uses the *Krypton* palette to obtain values for the drawing of the control.

Implementation
It is important to note that the implementation of the *Krypton* control is achieved by embedding a standard windows forms *MaskedTextBox* control inside a custom control. As such the functionality of the masked text box portion of the control is the same as that of the standard windows forms control. The *Krypton* implementation draws the border around the embedded control area and also allows for the display of *ButtonSpec* instances.

You can directly access the embedded control instance by using the *KryptonMaskedTextBox.MaskedTextBox* property. This property is not exposed at design time but can be accessed directly using code. In most scenarios you should not need to access the underlying control as the majority of methods and properties for the embedded masked text box are exposed directly by the *KryptonMaskedTextBox*. However if you need to implement drag and drop functionality you will need to hook directly into the events exposed by the embedded masked text box and not the drag and drop events of the *Krypton* custom control.

Appearance
The *InputControlStyle* property has a default value of *Standalone* giving the same appearance as seen in figure 1. As the name suggests this is intended for use in a scenario where the control is used in a standalone fashion and used on something like a *KryptonPanel* or *KryptonGroup*. The *InputControlStyle* of *Ribbon* is intended for use when the control is present inside the *KryptonRibbon* and then needs a different appearance and operation.

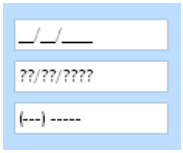


Figure 1 – InputControlStyle = Standalone

Three States
Only three possible states of *Disabled, Normal* or *Active* are used by the masked text box control. In order to customize the appearance use the corresponding *StateDisabled, StateNormal* and *StateActive* properties. Each of these properties allows you to modify the background, border and content characteristics. Note that the control is restricted to the *Disabled* and *Active* states if the *AlwaysActive* property is defined as *True*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

AlwaysActive
This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The top instance does not have the mouse over it and the bottom instance does.

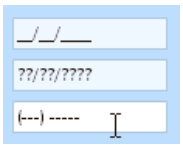


Figure 2 – InputControlStyle = Ribbon

ButtonSpecs
Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control. Figure 3 shows an example of a button specification that has been created to be positioned at the *Far* edge with a button style of *ButtonSpec* and a button type of *Context*. You could then use this button to show a context menu with additional options relevant to the entry field. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

Figure 3 – InputControlStyle = Ribbon

AllowButtonSpecTooltips
By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

KryptonMonthCalendar

The *KryptonMonthCalendar* control presents an intuitive graphical interface for users to view and set date information. The control displays a grid containing the numbered days of the month, arranged in columns underneath the days of the week. You can select a different month by clicking the arrow buttons on either side of the month caption. This control uses the *Krypton* palette to obtain values for the drawing of the control.

Appearance

The calendar displays with a border that contains a top and bottom caption with the numbered days in the center as can be seen in figure 1. The top caption is used to display the month and year of the date displayed along with arrows that can be used to navigate forward and backward a month at a time. At the bottom is a caption area that shows today's date which acts as a button so that pressing the date will shift the selection to the date displayed. In the center are six rows of days that display the possible days that can be selected. Note that days that are not part of the month are shown in a disabled color.

Figure 1 – Office 2007 Blue Palette

Seven States

There are seven state properties that can be used to modify the appearance for different elements of the control for the different possible display states. The control background, control border and day heading elements can be customized by using the *StateNormal* and *StateDisabled* set of properties. Individual numbered days can be customized using all seven sets of properties *StateNormal*, *StateDisabled*, *StateTracking*, *StatePressed*, *StateCheckedNormal*, *StateCheckedTracking* and *StateCheckedPressed*. The checked group of states are used for a day this is currently part of the selection. Tracking and Pressed states represent when the mouse is hovering over the day or when the left mouse button is pressed down for the day respectively.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

Three Overrides

Override properties are used when a particular condition has been met and are used in preference to the State property that would normally be used. The *OverrideFocus* values are used if the control currently has focus allows you to specify how to modify the appearance of a numbered day when it has focus. By default the focus override only alters the appearance so that a focus rectangle is drawn around the numbered day so that the user can see that the control currently has the focus.

The *OverrideBolded* is used for a numbered day that needs to be more prominent because that day has been specified by the *AnnuallyBoldedDates*, *MonthlyBoldedDates* or *BoldedDates* collection properties. As the name suggests, by default a matching day will be shown in bold instead of the regular font. But you can modify the *OverrideBolded* settings to change the feedback and provide a different background or foreground color instead.

Finally the *OverrideToday* is used to alter the appearance of the day that is defined as today's date. You can change *TodayDate* property if you need to specify a particular date as today. By default the control will draw the border of the numbered date that matches *TodayDate* in a different color. Figure 1 shows an example where the day has a dark red border color.

ButtonSpecs

Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the bottom caption. Figure 2 shows an example of a button specification that has been created to be positioned at the *Far* edge with a button style of *ButtonSpec* and a button type of *Context*. You could then use this button to show a context menu with additional options relevant to the calendar. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

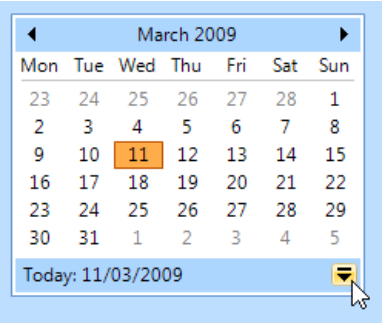


Figure 2 – KryptonMonthCalendar with ButtonSpec definition

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

DataBinding

There are three properties that make good candidates for data binding against. The first is called *SelectionRange* and is named the same as the property found on the standard windows *MonthCalendar* control. Unfortunately this is hard to use because it consists of two child values for the start and end of the range. Instead we recommend that you actually bind against the *SelectionStart* and *SelectionEnd* properties. These are both of type *DateTime* and are simple to bind against.

KryptonNumericUpDown

The *KryptonNumericUpDown* control allows you to display numeric values with the ability to use spin buttons to change the value. You can define value limits to prevent the user from entering invalid values and also decide if decimal places or thousand separators are required. This control uses the *Krypton* palette to obtain values for the drawing of the control.

Implementation

It is important to note that the implementation of the *Krypton* control is achieved by embedding a standard windows forms *NumericUpDown* control inside a custom control. As such the functionality of the edit portion of the control is the same as that of the standard windows forms control. The *Krypton* implementation draws the border around the embedded control area and also allows for the display of *ButtonSpec* instances.

You can directly access the embedded control instance by using the *KryptonNumericUpDown.NumericUpDown* property. This property is not exposed at design time but can be accessed directly using code. In most scenarios you should not need to access the underlying control as the majority of methods and properties for the embedded numeric control are exposed directly by the *KryptonNumericUpDown*. However if you need to implement drag and drop functionality you will need to hook directly into the events exposed by the embedded control and not the drag and drop events of the *Krypton* custom control.

Appearance

The *InputControlStyle* property has a default value of *Standalone* giving the same appearance as seen in figure 1. As the name suggests this is intended for use in a scenario where the control is used in a standalone fashion and used on something like a *KryptonPanel* or *KryptonGroup*. The *InputControlStyle* of *Ribbon* is intended for use when the control is present inside the *KryptonRibbon* and then needs a different appearance and operation.

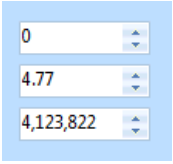


Figure 1 – InputControlStyle = Standalone

Three States

Only three possible states of *Disabled, Normal* or *Active* are used by the numeric control. In order to customize the appearance use the corresponding *StateDisabled, StateNormal* and *StateActive* properties. Each of these properties allows you to modify the background, border and content characteristics. Note that the control is restricted to the *Disabled* and *Active* states if the *AlwaysActive* property is defined as *True*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

AlwaysActive

This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The top instance does not have the mouse over it and the bottom instance does.

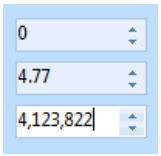


Figure 2 – InputControlStyle = Ribbon

ButtonSpecs

Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control. Figure 3 shows an example of a button specification that has been created to be positioned at the *Far* edge with a button style of *ButtonSpec* and a button type of *Context*. You could then use this button to show a context menu with additional options relevant to the entry field. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

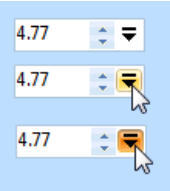


Figure 3 – KryptonNumericUpDown with ButtonSpec definition

AllowButtonSpecTooltips
By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

KryptonPanel
KryptonPanel

Use the *KryptonPanel* control when you need to provide an identifiable area for other controls. For example, you can use the panels to subdivide a form into distinct areas. Moving the panel will cause all the contained controls to also be moved along with it as it acts as a container. This control is similar to the *KryptonGroup* except it provides only a background and whereas the *KryptonGroup* also draws a border.

The *KryptonPanel* is more suited towards providing the background for large sections of the client area. *KryptonGroup* is more suitable for grouping a small number of related controls together.

Appearance
The *PanelBackStyle* property is used to define the top level styling required for the background appearance of the *KryptonPanel* control. The default value of *PanelClient* gives a background appropriate for use filling the client area of a form. Alternatively use the *PanelAlternate* setting for a panel that needs to stand out. See figures 1 and 2 for examples of the visual difference. There is also a custom style that can be defined via a *KryptonPalette* for situations where you need to create a variation on the styles already provided. The custom style is called simply *Custom1*.

Two States
Only two possible states of *Disabled* and *Normal* are used by the panel control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Note that only the background characteristics can be modified as the panel control never has a border or content.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the background color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Examples of Appearance
Figure 1 shows the appearance when the default *PanelBackStyle* of *ClientPanel* is used. You can see that the *Disabled* and *Normal* states have the same appearance. This is true for all the *PanelBackStyle* values as the panel does not give any visual indication of the enabled state of the control. If you need to show the panel is disabled then you can alter the *StateDisabled* property as required. Figure 2 shows *PanelBackStyle* with a value of *PanelAlternate*.

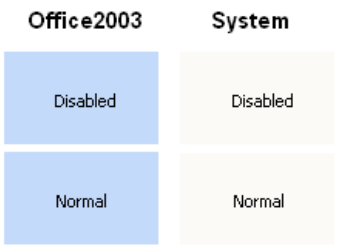


Figure 1 – PanelBackStyle = PanelClient

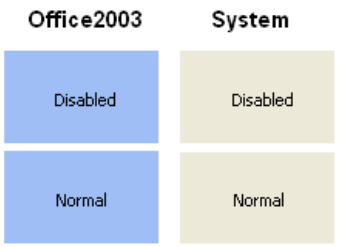


Figure 2 – PanelBackStyle = PanelAlternate

KryptonRadioButton

Multiple *KryptonRadioButton* instances are used together to provide a group of possible options that the user can choose from. This control combines the radio button functionality with the styling features of the *Krypton Toolkit*. The user can check the radio button by using the mouse or a keyboard mnemonic. Place code in the *CheckedChanged* event handler to perform actions based on the checked state. The content of the control is contained in the *Values* property. You can define *Text*, *ExtraText* and *Image* details within the *Values* property.

Appearance
The control uses a label style for presenting the text and image associated with the radio button. The *LabelStyle* property has a default value of *NormalControl* giving the same appearance as a *KryptonLabel* instance. If you place the radio button control onto a *Panel* background then you are recommended to change the *LabelStyle* to *NormalPanel* so that the label has an appropriate contrasting appearance.

You can use the *Orientation* property to rotate the control. The default setting of *Top* shows the content in a left to right and top to bottom arrangement. Specify *Bottom* to have the control displayed upside down, *Left* to show the content rotated 90 degrees left and *Right* for 90 degrees rotated right. See figure 1 for examples.

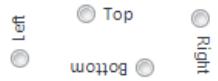


Figure 1 – Orientation Property

To alter the relative location of the radio button use the *CheckPosition* property. The default is Left and shows the radio button on the left side of the control values. Alternatively use the *Top*, *Bottom* or *Right* values as shown in Figure 2.

Figure 2 – Orientation Property

Checked
The Checked property is a boolean property that returns *True* when the radio button is checked and otherwise returns *False*. You can hook into the *CheckedChanged* event in order to be notified when the property has changed values.

AutoCheck
If the radio button becomes checked you would normally want all other radio buttons inside the same container to become unchecked. With this property defined as *True*, which is the default, this will happen automatically. If instead you would prefer to manually update the checked state of your radio buttons then set this to *False* for all instances inside the same container.

Two States
Only two possible states of *Disabled* and *Normal* are used by the radio button control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Note that only the content characteristics can be modified as the radio button never has a border or background.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the text font in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Focus Override
If the control currently has the focus then the *OverrideFocus* settings are applied. This can occur when the control is in the *Normal* state. By default the override only alters the appearance so that a focus rectangle is drawn around the content so that the user can see that the control currently has the focus.

Examples of Appearance
Figure 3 shows the appearance when using the *Office 2007 - Blue* palette with the default settings. You can see checked and unchecked appearance when disabled, normal, tracking or pressed down.

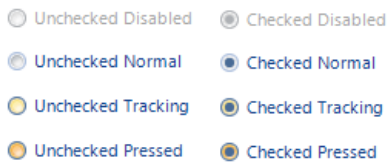


Figure 3 – RadioButton Appearance

KryptonRichTextBox

The *KryptonRichTextBox* control is used for displaying, entering and manipulating text with formatting. The control does everything the KryptonTextBox control does, but it can also display fonts, colors and links; load text and embedded images from a file; undo and redo editing operations; and find specified characters. This control uses the *Krypton* palette to obtain values for the drawing of the control.

Implementation
It is important to note that the implementation of the *Krypton* control is achieved by embedding a standard windows forms *RichTextBox* control inside a custom control. As such the functionality of the rich text box portion of the control is the same as that of the standard windows forms control. The *Krypton* implementation draws the border around the embedded control area and also allows for the display of *ButtonSpec* instances.

You can directly access the embedded control instance by using the *KryptonRichTextBox.RichTextBox* property. This property is not exposed at design time but can be accessed directly using code. In most scenarios you should not need to access the underlying control as the majority of methods and properties for the embedded rich text box are exposed directly by the *KryptonRichTextBox*. However if you need to implement drag and drop functionality you will need to hook directly into the events exposed by the embedded rich text box and not the drag and drop events of the *Krypton* custom control.

Appearance
The *InputControlStyle* property has a default value of *Standalone* giving the same appearance as seen in figure 1. As the name suggests this is intended for use in a scenario where the control is used in a standalone fashion and used on something like a *KryptonPanel* or *KryptonGroup*. The *InputControlStyle* of *Ribbon* is intended for use when the control is present inside the *KryptonRibbon* and then needs a different appearance and operation.

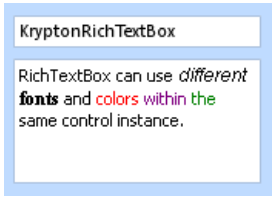


Figure 1 – InputControlStyle = Standalone

Three States
Only three possible states of *Disabled, Normal* or *Active* are used by the rich text box control. In order to customize the appearance use the corresponding *StateDisabled, StateNormal* and *StateActive* properties. Each of these properties allows you to modify the background, border and content characteristics. Note that the control is restricted to the *Disabled* and *Active* states if the *AlwaysActive* property is defined as *True*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

AlwaysActive
This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The top instance does not have the mouse over it and the bottom instance does.

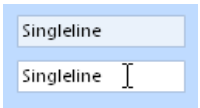


Figure 2 – InputControlStyle = Ribbon

ButtonSpecs
Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control. Figure 3 shows an example of a button specification that has been created to be positioned at the *Far* edge with a button style of *ButtonSpec* and a button type of *Context*. You could then use this button to show a context menu with additional options relevant to the entry field. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

Figure 3 – InputControlStyle = Ribbon

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

KryptonSeparator

This simple control is used to display an area separator that can act as a splitter. Press down with the left mouse button and then start dragging and an overlay will be shown indicating where the separator will move if you release the mouse at that position. This is ideal for use between controls that you want the user to be able to resize easily. The *KryptonSeparator* itself does not perform the resizing of the sibling controls but instead generates events that you can hook into and use to perform the appropriate change yourself.

Appearance

The *SeparatorStyle* property is used to define the styling required for the *KryptonSeparator* control. The most commonly used styles are *HighProfile* and *LowProfile*. *HighProfile* will shown a visual representation of a splitter with a handle so that users can easily see that they should drag the area to change relative sizing. *LowProfile* should be used when you want the background style to show through as defined by the *ContainerBackStyle* property.

You can use the *Orientation* property to control the drawing direction of the separator. The default setting of *Vertical* shows the content drawn as if the separator is used between two horizontal items and the separator is a thin splitter with a high vertical size but slim horizontal size. Specify *Horizontal* to have the control in the opposite orientation. See figure 1 for examples.

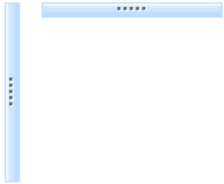


Figure 1 – Orientation Property

Four States

The separator control has four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. If the control has been disabled because the *Enabled* property is defined as *False* then the separator will be in the *Disabled* state.

When enabled the separator will be in the *Normal* state until the user moves the mouse over the splitter area at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the control then it enters the *Pressed* state. In order to customize the appearance of the control in each of the four states you can use the properties *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed*. Each of these properties allows you to modify the background of the control and the background, border and padding of the inner splitter element.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the background color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Examples of Appearance

Figure 2 shows the control appearance and the mouse cursor during a dragging operation. The top image has the separator without the mouse over the control. Next image down is the display when the mouse moves over the separator and you can see the cursor has been updated to indicate the dragging is possible. Last is the display you see whilst the mouse is pressed and being dragged, you can see the black box that provides feedback on the change that will occur if the mouse is released in that position.

Figure 2 – Example appearance during operation

Events

SplitterMoveRect

This event is fired when the use presses down on the separator and is sent to request the size of the area that the splitter is allowed to be moved. You should update the *MoveRect* property of the event arguments to set the bounding box that the splitter indication is allowed to be moved to. You can cancel this event if you decide the action should not be allowed.

SplitterMoving

As the mouse is moved during the drag this event is fired to inform you of the new position.

SplitterMoved

SplitterNotMoved

When the drag operation completes with success the *SplitterMoved* is fired so you can perform what resizing operation is appropriate. If the drag operation is abandoned the *SplitterNotMoved* is fired instead so you can cleanup any actions you performed ni the *SplitterMoveRect*.

KryptonSplitContainer

Use the *KryptonSplitContainer* to control the sizing of two panels. A movable bar sits between the two panels and allows the user to change the relative spacing. You can create complex user interfaces by nesting controls within each of the two panels. The panels are derived from the *KryptonPanel* control.

Appearance
The *ContainerBackStyle* property is used to define the top level styling required for the background of the container that excludes the splitter area. Use the *SeparatorStyle* property to alter the styling of the splitter area. Under normal circumstances the *ContainerBackStyle* will not have any visual affect because the two panels are positioned over the affected area. Only if the panels are modified to have a transparent background will this styling property become useful.

You can access the two panel instances directly by using the *Panel1* and *Panel2* properties. At design time you can drag and drop controls directly onto the panels in order to setup the required appearance.

Use the *Orientation* property to alter the splitter from working in a vertical to horizontal manner. See figure 1 for an example of the difference as seen at design time.

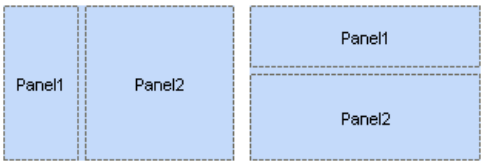


Figure 1 – Vertical and Horizontal orientation

Layout
FixedPanel can be used to indicate that one of the panels should remain a constant size and only the other panel should be resized when the split container is resized. Otherwise the default is to resize both panels to maintain the same relative sizes. *IsSplitterFixed* is *False* by default and so allows the user to drag the splitter in order to alter the relative sizes of the panels. Set this to *True* in order to prevent the user from making any change.

To enforce a minimum size use the *Panel1MinSize* and *Panel2MinSize* properties. This is useful in preventing the user from resizing a panel down to zero or very small sizes that would be inappropriate in your application. To hide one of the panels from display use *Panel1Collapsed* or *Panel2Collapsed* as required. When a panel is collapsed the other panel will automatically take up the entire split container area and the splitter removed from view.

SplitterDistance is the number of pixels from the left or top of the split container that the splitter should be positioned. When the *Orientation* is vertical the distance is from the left and when horizontal the distance is measured from the top. *SplitterWidth* is the pixel thickness of the splitter area. *SplitterIncrement* is pixel multiple used for moving the splitter position. So an increment of 10 pixels would mean the user can only move the splitter in 10 pixel increments from the current position.

Events
When the splitter position is being moved the *SplitterMoving* event is generated and allows the change to be cancelled. Once the move has been completed the *SplitterMoved* event is fired.

Four States
The split container can be in one of four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. If the control has been disabled because the *Enabled* property is defined as *False* then the button will be in the *Disabled* state.

When enabled the split container will be in the *Normal* state until the user moves the mouse over the splitter area at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the splitter then it enters the *Pressed* state.

In order to customize the appearance of the control in each of the four states you can use the properties *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed*. Each of these properties allows you to modify the background of the split container and then background and border of the splitter area.

Common State
To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the background color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the background of the splitter area to be always red. Without the *StateCommon* property you would need to update the same setting in each of *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed* properties. Instead you can define the setting in just *StateCommon* and know they will be used whichever of the four

appearance states the button happens to be using.

KryptonTextBox

The *KryptonTextBox* control allows the user to enter text or is used to present text. The control is generally used for editable text although it can be made read-only. It can display multiple lines, wrap text to the size of the control and add basic formatting. This control uses the *Krypton* palette to obtain values for the drawing of the control.

Implementation
It is important to note that the implementation of the *Krypton* control is achieved by embedding a standard windows forms *TextBox* control inside a custom control. As such the functionality of the text box portion of the control is the same as that of the standard windows forms control. The *Krypton* implementation draws the border around the embedded control area and also allows for the display of *ButtonSpec* instances.

You can directly access the embedded control instance by using the *KryptonTextBox.TextBox* property. This property is not exposed at design time but can be accessed directly using code. In most scenarios you should not need to access the underlying control as the majority of methods and properties for the embedded text box are exposed directly by the *KryptonTextBox*. However if you need to implement drag and drop functionality you will need to hook directly into the events exposed by the embedded text box and not the drag and drop events of the *Krypton* custom control.

Appearance
The *InputControlStyle* property has a default value of *Standalone* giving the same appearance as seen in figure 1. As the name suggests this is intended for use in a scenario where the control is used in a standalone fashion and used on something like a *KryptonPanel* or *KryptonGroup*. The *InputControlStyle* of *Ribbon* is intended for use when the control is present inside the *KryptonRibbon* and then needs a different appearance and operation.

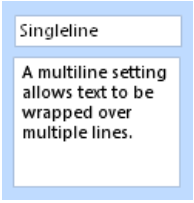


Figure 1 – InputControlStyle = Standalone

Three States
Only three possible states of *Disabled, Normal* or *Active* are used by the text box control. In order to customize the appearance use the corresponding *StateDisabled, StateNormal* and *StateActive* properties. Each of these properties allows you to modify the background, border and content characteristics. Note that the control is restricted to the *Disabled* and *Active* states if the *AlwaysActive* property is defined as *True*.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon.*

AlwaysActive
This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The top instance does not have the mouse over it and the bottom instance does.

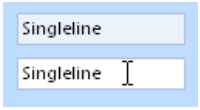


Figure 2 – InputControlStyle = Ribbon

ButtonSpecs
Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control. Figure 3 shows an example of a button specification that has been created to be positioned at the *Far* edge with a button style of *ButtonSpec* and a button type of *Context*. You could then use this button to show a context menu with additional options relevant to the entry field. Other possible uses of button specifications might be to indicate error conditions or to initiate the showing of help information.

Figure 3 – InputControlStyle = Ribbon

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

KryptonTrackBar

The *KryptonTrackBar* is a scrollable control similar to the scroll bar. You can configure ranges through which the value of the *Value* property of a track bar scrolls by setting the *Minimum* property to specify the lower end and the *Maximum* property to specify the upper end of the range. The *LargeChange* property defines the increment to add or subtract from the *Value* property when clicks occur either side of the track position indicator. The *SmallChange* property defines the increment to add or subtract from the *Value* property when using the keyboard. You can use the *Orientation* property to change the default horizontal appearance to become vertical.

Appearance

The *Orientation* property is used to change the direction of the track bar control. Figure 1 shows the vertical and horizontal values. You will see also in Figure 1 that ticks marks are displayed on both sides of the track. You can use the *TickStyle* to modify this and present tick marks on only one of the sides or not at all. Use the *TickFrequency* property to define how often a tick mark is drawn. By default the tick marks are drawn for each value between the *Minimum* and *Maximum* property values but you can alter the *TickFrequency* to show them less often. Finally the *TrackBarSize* is an enumeration property with possible values of *Small*, *Medium* and *Large* and specifies how large the track and track indicator are drawn. Figure 1 shows instances with the default value of *Medium*.

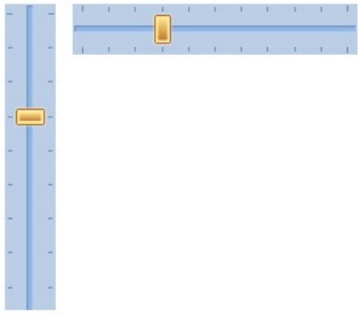


Figure 1 - Orientation settings.

Four States

The track bar can be in one of four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. If the control has been disabled because the *Enabled* property is defined as *False* then the track bar will be in the *Disabled* state. When enabled the track bar will be in the *Normal* state until the user moves the mouse over the track bar position indicator at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the position indicator then it enters the *Pressed* state.

In order to customize the appearance of the control in each of the four states you can use the properties *StateDisabled*, *StateNormal*, *StateTracking* and *StatePressed*.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied. This can occur when the control is in *Normal*, *Tracking* or *Pressed* states. By default the override only alters the appearance so that the track bar position indicator shows that the control has the focus.

Examples of Appearance

Figure 2 shows the appearance when a track bar is using the *Office 2010 Blue* palette and for each of the different possible states.

Figure 2 - TrackBar states.

KryptonTreeView

Use the *KryptonTreeView*when you need to present a hierarchical collection of labeled items. If the number of items exceeds the number that can be displayed then a scroll bar is automatically shown. Use the *Nodes* property to define the top level root nodes of the hierarchy. Hook into the *AfterSelect* event to be notified when the selected node is changed.

Appearance

Use the *BackStyle*, *BorderStyle* and *ItemStyle* properties to alter the appearance of the control and the node items. The first two of these properties define the appearance of the overall control but the *ItemStyle* is used to define the display of the individual node items themselves. You can see in figure 1 the default appearance using the *Office 2010 - Blue* builtin palette.

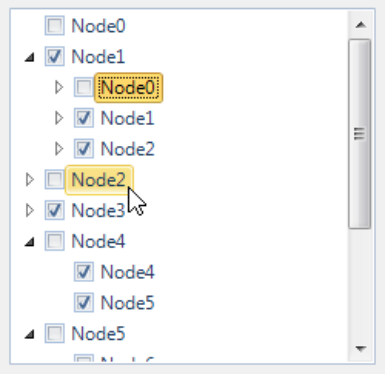


Figure 1 – Default Appearance

Eight States

There are eight different states relating to the control but not all the states are relevant to every part of the control.

The border and background of the control use just the *StateNormal*, *StateDisabled* or *StateActive* sets of properties. If the control has been disabled because the *Enabled* property is defined as *False* then the control always uses the *StateDisabled* values. When the control is active the *StateActive* properties are used and when not active the *StateNormal*. Being currently active means it has the focus or the mouse is currently over the control. Note that if the *AlwaysActive* property is defined as *True* then it ignores the *StateNormal* and always uses the *StateActive* regardless of the current active state of the control.

Drawing of the individual node items uses all the state properties except *StateActive*. This is because the node items are not drawn differently depending on the active state of the control. All the states that begin with the *StateChecked* name are used for items that are currently selected. When not selected an item uses one of the *StateNormal*, *StateTracking* or *StatePressed* properties.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Imagine the following scenario; you would like to define the border of the button to be 3 pixels wide with a rounding of 2 pixels and always red. Without the StateCommon property you would need to update the same three border settings in each of *StateDisabled*, *StateNormal*, *StateTracking* etc properties. Instead you can define the three border settings in just *StateCommon* and know they will be used whichever of the states the tree view happens to be using.

Focus Override

If the control currently has the focus then the *OverrideFocus* settings are applied to the item that has the focus indication. This can occur when the control is in *Normal*, *Tracking* or *Pressed* states. By default the override only alters the appearance so that a focus rectangle is drawn around the node item contents so that the user can see that the control currently has the focus.

AlwaysActive

This property is used to indicate if the control should always be in the active state. For an *InputControlStyle* of *Standalone* the default value of *True* is appropriate. However, when you switch to using the *InputControlStyle* of *Ribbon* you should alter this to *False*. A value of *False* means that when the mouse is not over the control and it also does not have focus it will be considered inactive. This allows you to specify a different appearance for the active and inactive states. Figure 2 shows an example of the *Ribbon* style with the *AlwaysActive* property defined as *False*. The left instance does not have the mouse over it and the right instance does.

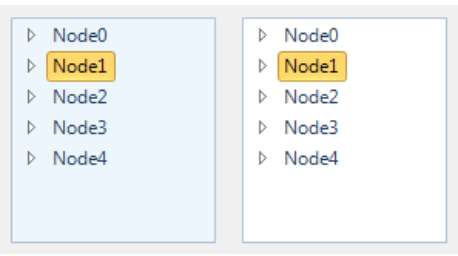


Figure 2 – InputControlStyle - Ribbon

KryptonTreeNode

You can add standard *TreeNode* instances to the node hierarchy and it will use the *Text* property to recover the text for display. Alternatively you can add *KryptonTreeNode* instances that derive from that standard *TreeNode* class and add an additional *LongText* property for display to the right of the standard node text. Figure 3 shows an example with several instances of this class added to the root *Nodes* collection.

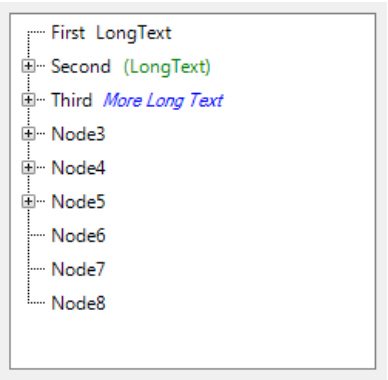


Figure 3 – Nodes collection containing KryptonTreeNode instances

KryptonWrapLabel

Use the *KryptonWrapLabel* control when you need to functionality of the standard *Windows.Forms.Label* control but with the text color and font of the *Krypton* palette system. Unlike the *KryptonLabel* control this *KryptonWrapLabel* control inherits directly from the standard *Windows.Forms.Label* class and so it performs the same automatic wrapping of text onto multiple lines. This is the one circumstance for which this control is recommended instead of the *KryptonLabel* control. If you need text to span multiple lines then this is the control to use.

Appearance

The *LabelStyle* property is used to define the top level styling required for the appearance of the *KryptonWrapLabel* control. The four standard values are *NormalControl*, *TitleControl*, *NormalPanel* and *TitlePanel*. It is important to understand when to use which style in order to get the correct appearance when switching to different palettes.

When using a wrap label that is positioned with on a *Control* style background, such as *ControlClient* or *ControlAlternate*, then you should use the *NormalControl* or *TitleControl* styles. A good example would be placing wrap label instances in the client area of a *KryptonHeaderGroup,* as the header group has a default background style in the client area of *ControlClient*. If however you are placing wrap label instances onto a *KryptonPanel* then you should use the *NormalPanel* or *TitlePanel* styles. It is easy to forget to set the appropriate style because most of the builtin palettes have colors that look fine on either a *Control* or *Panel* style background. But if you use the *Office 2007 - Black* palette then it will fail to appear correctly as the colors are radically different. In that scenario a *NormalControl* on a *Panel* background would be invisible!

The *NormalControl* and *NormalPanel* styles give a standard text appearance appropriate for use as a caption for other controls. Alternatively use the *TitleControl/TitlePanel* settings for use as a section header where the text needs to be more prominent. There are also custom styles that can be defined via a *KryptonPalette* for situations where you need to create variations on the styles already provided. Custom styles are named simply *Custom1*, *Custom2* and *Custom3*.

Two States

Only two possible states of *Disabled* and *Normal* are used by the wrap label control. In order to customize the appearance use the corresponding *StateDisabled* and *StateNormal* properties. Note that only three properties are provided. You can only customize the text color, text font and rendering hint. These are also the only three values that are inherited from the palette that defines the default properties. Also note that the background color cannot be set as the background is always transparent and so shows the background of the parent control.

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the text font in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Custom Control

When writing your own custom controls that are working alongside the *Krypton* controls you may want to ensure that the look and feel of your custom control matches that of the *Krypton* components. There are two levels of integration that you can aim for. The first is to use the same palette of colors, font, widths and other metrics when drawing and sizing your custom control. This is the purpose of the Using IPalette article. Recovering these metrics is actually very simple and you can examine the *Custom Control using Palettes* sample project to see the principle in action.

Alternatively you might want to leverage the same rendering code that the *Krypton Toolkit* controls use. In this case follow the Using IRenderer article to understand how to render background, border and content elements within your custom control client area. Although a little more complicated this technique has the advantage of performing the hard work for you. You can use the renderer to draw vertical orientated text and elements with tiled images without the need to write the actual drawing code yourself.

Using IPalette

Monitoring the global palette

By default the *Krypton* controls use the global palette that is specified in the *KryptonManager* component. In order to ensure your own custom control is consistent with the rest of the *Krypton* controls you need to also make use the global palette setting. The following steps will take you through the process of adding this support.

1) Setting up

We need to begin by adding a private field to the custom control class that caches the current global palette setting.

```
private IPalette _palette;
```

Next we need to modify the constructor by adding the 4 blocks of code as shown below.

```
public MyUserControl()      {
    // (1) To remove flicker we use double buffering for drawing
    SetStyle(
        ControlStyles.AllPaintingInWmPaint |
        ControlStyles.OptimizedDoubleBuffer |
        ControlStyles.ResizeRedraw, true);

    InitializeComponent();

    // (2) Cache the current global palette setting
    _palette = KryptonManager.CurrentGlobalPalette;

    // (3) Hook into palette events
    if (_palette != null)
      _palette.PalettePaint += new EventHandler<PaletteLayoutEventArgs>(OnPalettePaint);

    // (4) We want to be notified whenever the global palette changes
    KryptonManager.GlobalPaletteChanged += new EventHandler(OnGlobalPaletteChanged);
}
```

The first block of code, with (1) in the comment, is used to reduce the flicker that occurs when re drawing a control. Setting the *AllPaintingInWmPaint* and *OptimizedDoubleBuffer* control styles will ensure that all drawing takes place in an off screen buffer before being bit blitted to the screen. The *ResizeRedraw*control style is set so that whenever the size of the control changes the control is automatically invalidated, otherwise you need to manually override the *OnResize* method and request the invalidate in code.

The second block of code, marked (2), calls a static property on the *KryptonManager* component in order to recover the current global *IPalette* interface. Using this property ensures we always get back an *IPalette* without needing to worry if a built in palette or custom palette has been specified. The result of the call is cached in our private field _palette for use when painting or processing related events.

Now that we have the *IPalette* we use code block (3) to hook into the *PalettePaint* event that is fired by the palette whenever a palette setting changes that requires a repaint. This can happen with the built in palettes when the user alters the display settings or at any time for custom palettes. We will implement the trivial event handler for this in the section step.
Finally we need block (4) to hook into a static event exposed by the *KryptonManager* called *GlobalPaletteChanged*. As the name suggest this is fired whenever the global palette changes. When this happens we need our event handler to cache the new palette. Step 2 below will implement the actual event handlers that we just hooked into.

2) Responding to changes

Our event handler for code block (3) is trivial, when notified about a change in the palette settings we just request the control be painted.

```
private void OnPalettePaint(object sender, PaletteLayoutEventArgs e)      {
    Invalidate();
}
```

Handling a change in global palette requires a few steps. Code block (5) is used to reverse block (3) from above. Then we update, at block (6), the internal field for later use. Code (7) establishes a hook into the *PalettePaint* event of the new palette and finally at (8) we request the control be re painted to show the change in appearance.

```
private void OnGlobalPaletteChanged(object sender, EventArgs e)      {
      // (5) Unhook events from old palette
    if (_palette != null)
      _palette.PalettePaint -= new EventHandler<PaletteLayoutEventArgs>(OnPalettePaint);
```

```
        // (6) Cache the new IPalette that is the global palette
        _palette = KryptonManager.CurrentGlobalPalette;
        // (7) Hook into events for the new palette
        if (_palette != null)
          _palette.PalettePaint += new EventHandler<PaletteLayoutEventArgs>(OnPalettePaint);

        // (8) Change of palette means we should repaint to show any changes
        Invalidate();      }
```

3) Tearing down

To finish the palette monitoring we need to override the *Dispose* method and unhook our outstanding events. This is because the events exist on static instances and so our control instance will not be garbage collected until the static objects themselves are no longer required. In the case of the *KryptonManager* this will be never and so our custom control instances will never be garbage collected if we do not unhook the events.
Code block (10) below is used to unhook from the cached palette instance and block (11) un hooks from the KryptonManager static event.

```
    protected override void Dispose(bool disposing)     {
        if (disposing)          {
            // (10) Unhook from the palette events
            if (_palette != null)               {
                _palette.PalettePaint -= new EventHandler<PaletteLayoutEventArgs>(OnPalettePaint);
                _palette = null;
            }

            // (11) Unhook from the static events, otherwise we cannot be garbage collected
            KryptonManager.GlobalPaletteChanged -= new EventHandler(OnGlobalPaletteChanged);
        }

        base.Dispose(disposing);
    }
```

Recovering palette details

Now that we have a reference to the palette we are going to use we can actually implement the painting method of the custom control. The actual use of the *IPalette* itself is very simple as can be seen in the following example.

```
    protected override void OnPaint(PaintEventArgs e)     {
        if (_palette != null)          {
            // (12) Calculate the palette state to use in calls to IPalette
            PaletteState state = Enabled ? PaletteState.Normal : PaletteState.Disabled;

            // (13) Get the background, border and text colors along with the text font
            Color backColor = _palette.GetBackColor1(PaletteBackStyle.ButtonStandalone, state);
            Color borderColor = _palette.GetBorderColor1(PaletteBorderStyle.ButtonStandalone, state);
            Color textColor = _palette.GetContentShortTextColor1(PaletteContentStyle.ButtonStandalone, state);
            Font textFont = _palette.GetContentShortTextFont(PaletteContentStyle.ButtonStandalone, state);

            // Fill the entire background of the control
            using (SolidBrush backBrush = new SolidBrush(backColor))
              e.Graphics.FillRectangle(backBrush, e.ClipRectangle);

            // Draw a single pixel border around the control edge
            using (Pen borderPen = new Pen(borderColor))
              e.Graphics.DrawPath(borderPen, path);

            // Draw control Text at a fixed position
            using (SolidBrush textBrush = new SolidBrush(textColor))
              e.Graphics.DrawString("Click me!", textFont, textBrush, Width / 2, Height / 2);
        }

        base.OnPaint(e);
    }
```

Each call to the *IPalette* in order to recover a color, font or other metric must provide a *PaletteState* enumeration value. This informs the palette of the appropriate value to return relative to the supplied state. Our control is only going to provide one of the two basic values, either *PaletteState.Normal* or *PaletteState.Disabled*. Code block (12) is used to determine which of these to use.
Block (13) is where we actually recover metrics from the palette. In this case we want the background color, border color, text color and text font in the style of a *ButtonStandalone*. You would of course change the provided style values to whatever is appropriate for your own circumstances. Note that in this simple example we are only recovering the first color and then using those colors in the subsequent drawing code. This will result in always drawing solid colors and will not provide the gradient effect you are familiar with elsewhere in *Krypton* controls.

By following the steps outlined here you should now be in a position to recover whatever values you require from the global palette and use them in your custom drawing code. You are recommended to look at the source code for the *Custom Control using Palettes* example project that is provided with the library. The sample shows a slightly more extensive demonstration.

Using IRenderer

Monitoring the global palette

In order to get access to a renderer you need to begin with a palette. The recommended way of getting an *IRenderer* reference is to call the *GetReference()* method on the *IPalette* interface. Therefore your custom control should begin by monitoring the global palette so that you can access the associated renderer appropriate for that global palette. You should follow the steps outlined in the Using IPalette article in order to add ability.

IRenderer Basics

In order to use the methods exposed by the *IRenderer* we need take a few basic setup steps. In particular we need to create four helper objects that are then used in your interaction with the renderer instance. To begin you should add the following fields to your custom control class.

```
private PaletteRedirect _paletteRedirect;
private PaletteBackInheritRedirect _paletteBack;
private PaletteBorderInheritRedirect _paletteBorder;
private PaletteContentInheritRedirect _paletteContent;
private IDisposable _mementoBack;
```

Now update your constructor in order to create the four instances in the following manner.

```
public MyUserControl()
{
    // ...your other constructor code...
    // (1) Create redirection object to the base palette
    _paletteRedirect = new PaletteRedirect(_palette);

    // (2) Create accessor objects for the back, border and content
    _paletteBack = new PaletteBackInheritRedirect(_paletteRedirect);
    _paletteBorder = new PaletteBorderInheritRedirect(_paletteRedirect);
    _paletteContent = new PaletteContentInheritRedirect(_paletteRedirect);
}
```

In our previous article Using IPalette you could see how palette values were directly recovered from the *IPalette* interface. However the renderer does not interact directly with *IPalette* but instead with a set of more specific interfaces such as *IPaletteBack*, *IPaletteBorder* and *IPaletteContent*. Code block (1) creates a helper object that exposes these specific interfaces and redirects the *IPaletteBack, IPaletteBorder, IPaletteContent* calls into *IPalette* requests.
We then use block (2) to create helper objects for each of the specific interfaces. These are the actual objects you will pass into the renderer requests. They each have a *Style* property that allows you to specify the drawing style you require and all calls into the helper object will automatically be passed on with the style requested. You can see how this works in the code below. The following code is the template you should use in your drawing code.

```
protected override void OnPaint(PaintEventArgs e)
{
    if (_palette != null)
    {
        // (3) Get the renderer associated with this palette
        IRenderer renderer = _palette.GetRenderer();
        // (4) Create the rendering context that is passed into all renderer calls
        using (RenderContext renderContext = new RenderContext(this, e.Graphics, e.ClipRectangle, renderer))
        {
            // (5) Set style required when rendering
            _paletteBack.Style = PaletteBackStyle.ButtonStandalone;
            _paletteBorder.Style = PaletteBorderStyle.ButtonStandalone;
            _paletteContent.Style = PaletteContentStyle.ButtonStandalone;

            // (6) ...perform renderer operations...
        }       }
}
```

Code block (3) is used to grab the renderer that is associated with the palette we are using. We do not cache the renderer reference as we have already cached and tracked the global palette, so we only need to request the renderer when it is needed. Block (4) is used to create an instance of the *RenderContext* type that is passed into all renderer calls. We use this object in order to package up common values and so reduce the number of parameters passed into each renderer call.
In order to specify the drawing styles that should be used when recovering palette values we use block (5). In our example we are

setting these values in every call to the *OnPaint* call as an illustration. In your own application you might prefer to set these values just once in the constructor because you know the style values are not going to change. On the other hand you might need to alter the style settings between renderer calls in order to draw different styles of elements within your custom control.
Finally comment (6) shows the placeholder location where you would insert your rendering calls. The following sections of the article give details of how to make use of the background, border and content abilities of the renderer.

Drawing a Background

To draw a background we use the *IPalette.RenderStandardBack.DrawBack* call. The following code provides you will a example usage.

```
// (7) Do we need to draw the background?
if (_paletteBack.GetBackDraw(PaletteState.Normal) == InheritBool.True)
{
    using (GraphicsPath path = new GraphicsPath())
    {
        // (8) Add entire control client area to drawing path
        path.AddRectangle(ClientRectangle);

        // (9) Perform drawing of the background clipped to the path
        _mementoBack = renderer.RenderStandardBack.DrawBack(renderContext,
                                                            ClientRectangle,
                                                            path,
                                                            _paletteBack,
                                                            VisualOrientation.Top,
                                                            PaletteState.Normal,
                                                            _mementoBack);
    }
}
```

First we test to ensure that the palette specifies that the background is allowed to be drawn for the state we are requesting, as can be seen in code block (7). In this and all other examples in the article we provide the fixed *PaletteState.Normal* enumeration value. You should however provide the appropriate state for your control. Even the simplest control will need to provide one of two possible values, *PaletteState.Normal* when enabled and PaletteState.Disabled otherwise. If you need to provide more extensive feedback then you should consider supplying *PaletteState.Tracking* when the house is over the rendering element and *PaletteState.Pressed* when the mouse is also pressed down.
Code block (8) completes the process of creating a *GraphicsPath* that is used to clip the area of background drawn. In our case we want to draw the entire client area of the control and so set the path to the *ClientRectangle*, but you could create any path you like in order render complex shapes.
Finally block (9) performs the actual rendering operation. The second parameter is the rectangle you would like to be drawn and the third parameter a path used to clip the drawing operation. The second to last parameter is the palette state that you should ensure is the same as the call in block (7). The last parameter provides a memento object that is also assigned to as the result of the call. This technique allows performance improvements in the renderer as it allows the renderer to create caching objects and have them supplied again on subsequent calls.

Drawing a Border

Once you know how to draw a background the border drawing is trivial. Here is the code.

```
// (10) Do we need to draw the border?
if (_paletteBorder.GetBorderDraw(PaletteState.Normal) == InheritBool.True)
{
    // (11) Draw the border inside the provided rectangle area
    renderer.RenderStandardBorder.DrawBorder(renderContext,
                                             ClientRectangle,
                                             _paletteBorder,
                                             VisualOrientation.Top,
                                             PaletteState.Normal);
}
```

As with drawing the background we being with block (10) which tests to ensure we need to draw the border at all. If we do then we just need to make the call to the *IRenderer.RenderStandardBorder.DrawBorder* with a set of simple parameters, as seen in code block (11). The second parameter is the rectangle that specifies the outside of the border drawing area, the border itself will be drawn to fit completely within this rectangle.

Drawing a Background/Border Pair

In many cases you will be drawing a background and border for the same visual element. So if you control is drawing a button element in the client area then you would want to draw the button area background followed by the button area border. You might think this is achieved by performing the above two sections of code for the same rectangle area. In practice this is not quite the case because of a feature with borders.
It is possible for a border to have rounded corners and so we cannot just draw the background as a rectangle, if we did that then the background would be drawn outside the rounded corners. In order to prevent this we use a slightly modified version of the background drawing code presented above.

```
// Do we need to draw the background?
if (_paletteBack.GetBackDraw(PaletteState.Normal) == InheritBool.True)
{
    // (12) Get the background path to use for clipping the drawing
    using (GraphicsPath path = renderer.RenderStandardBorder.GetBackPath(renderContext,
                                                                          ClientRectangle,
                                                                          _paletteBorder,
                                                                          VisualOrientation.Top,
                                                                          PaletteState.Normal))
    {
        // Perform drawing of the background clipped to the path
        _mementoBack = renderer.RenderStandardBack.DrawBack(renderContext,
                                                            ClientRectangle,
                                                            path,
                                                            _paletteBack,
                                                            VisualOrientation.Top,
                                                            PaletteState.Normal,
                                                            _mementoBack);
    }
}
```

If you look at code block (8) from the previous drawing code then you will note that the clipping path for the background is specified by using a rectangle. Instead we use block (12) above to create a clipping path that is a description of the border. This prevents the background being drawn outside of the border. It uses a call to the *IRenderer.RenderStandardBorder.GetBackPath* to get the clipping path.

Drawing a Content

All *Content* consists of providing three values, two text strings and an image. In order to provide these values to the renderer an interface is used called *IContentValues*. You can implement this interface in any way that is appropriate for your usage but in our example we will choose the simplest option and implement it on the custom control itself. If you choose to do that same then you first of all need to add it to the class definition in the following way.

```
public class MyUserControl : UserControl, IContentValues
```

Next the actual interface methods need to be implemented, our examples just returns some simple fixed values.

```
public Image GetImage(PaletteState state)     {
    return null;
}

public Color GetImageTransparentColor(PaletteState state)     {
    return Color.Empty;
}

public string GetLongText()     {
    return "Click me!";
}

public string GetShortText()     {
    return string.Empty;
}
```

Unlike the background and border rendering we need to override the control layout event *OnLayout*. We do this so that the renderer can calculate the position and visibility of the content values and cache the results for when painting later. A control will usually be painted much more often than it is layed out and so caching the results of the layout event gives better performance than doing the calculations during every paint cycle.
In order to cache the results of the *OnLayout* we need to add a new field to the control as follows.

```
private IDisposable _mementoContent;
```

The actual *OnLayout* method is very similar to the way that the *OnPaint* works, recovering the *IRenderer* instance and then creating a context object that is passed into each of the renderer calls. The following implementation caches the result of laying out the content.

```
protected override void OnLayout(LayoutEventArgs e)     {
```

```
        if (_palette != null)
        {
            // Get the renderer associated with this palette
            IRenderer renderer = _palette.GetRenderer();

            // Create a layout context used to allow the renderer to perform layout
            using (ViewLayoutContext viewContext = new ViewLayoutContext(this, renderer))
            {
                // (13) Cleaup resources by disposing of old memento instance
                if (_memento != null)                    _memento.Dispose();

                // (14) Ask the renderer to work out how the Content values will be layed out
                // and return a memento object that we cache for use when performing painting
                _mementoContent = renderer.RenderStandardContent.LayoutContent(viewContext,
                                                                                ClientRectangle,
                                                                                _paletteContent,
                                                                                this,
                                                                                VisualOrientation.Top,
                                                                                false,
                                                                                PaletteState.Normal);
            }
        }

        base.OnLayout(e);
    }
```

Block (13) disposes of any existing memento instance to ensure that any resources that are held by the memento are correctly released.
Block (14) is the actual call to the renderer and parameter four provides the reference to the *IContentValues* interface that provides the content values. As the interface is implemented by the custom control in our example the reference is the *this* variable. As with all the other calls to the renderer you need to provide the appropriate palette state in place of the *PaletteState.Normal* constant in the examples.
Finally the actual *OnPaint* code needed to draw the content is presented.

```
    // Do we need to draw the content?
    if (_paletteContent.GetContentDraw(PaletteState.Normal) == InheritBool.True)
    {
        // Draw content using the memento cached from OnLayout
        renderer.RenderStandardContent.DrawContent(renderContext,
                                                   ClientRectangle,
                                                   _paletteContent,
                                                   _memento,
                                                   VisualOrientation.Top,
                                                   false,
                                                   PaletteState.Normal);
    }
```

As always the code checks that the content needs to be drawn all given the provided palette state.
Using the above information and examples it should be possible to experiment and draw whatever elements you need in your custom control in order to leverage the functionality provided in the toolkit as well as remaining faithful to the global palette settings. You should refer to the *Custom Control using Renderers* example project for a working example of the renderer code in action.

KryptonCheckSet

Use the *KryptonCheckSet* component when you need to group together several *KryptonCheckButton* instances in order to enforce the rule that only a single check button is checked at a time. So when the user clicks a check button it becomes checked and all others in the group are unchecked. Using this component removes the need to implement the functionality manually using event handlers attached to each of the check button controls.

The *CheckButtons* property is a collection of *KryptonCheckButton* references that are to be managed by the component. At design time this collection can be easily edited by using the collection editor that appears when the property button is pressed.

To specify the check button that is currently checked within the set you can use either the *CheckedButton* or *CheckedIndex* property. At design time you are recommended to use the *CheckedButton* property as the drop down list it presents gives the available options making selection easier.

AllowUncheck can be used to specify if the checked button when clicked becomes unchecked or is left unchanged. By default the property is *False* and so clicking the same check button multiple times has effect.

Whenever the currently checked button changes the *CheckedButtonChanged* event is fired so you can hook into the event to perform selection specific actions.

KryptonCommand

Use the *KryptonCommand* component to simplify the management of application actions.

For example, you could use a *KryptonCommand* instance to manage the state of your applications *Print* action. Your application might have three separate places where feedback is provided to the user about the availability of the *Print* capability. Instead of updating all three controls each time the *Print* state changes you only need to update the single command. The three attached controls will then be updated automatically. Just assign the command instance to the *KryptonCommand* property of any compatible control and the control will automatically keep itself in sync with the command state.
The command also exposes an event called *Execute* that you should hook into for notification of when the command needs to be executed. When the user presses a button that is attached to a command the button will cause the *Execute* event to be fired. So management of your *Print* action could not be easier. Just attach the command to your *KryptonButton*, *KryptonCheckButton*, *ButtonSpec*, *KryptonRibbon* element etc and those controls will automatically show the correct command state for you. Then when the user presses the *KryptonButton*, *ButtonSpec* etc the *Execute* event will fire so you can process the *Print* request.

KryptonCommand Properties

Figure 1 shows the list of properties exposed by the *KryptonCommand* component.

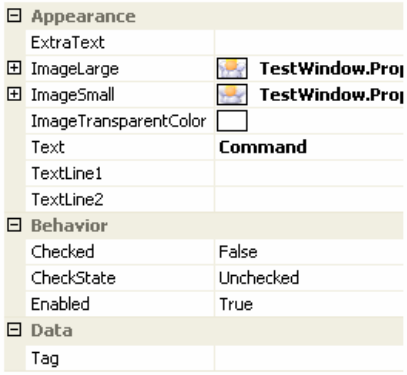


Figure 1 - KryptonCommand properties

ExtraText
This is the supplementary text used by attached controls for display.

ImageLarge ImageSmall
Most attached controls will use the *ImageSmall* property value but the *KryptonRibbon* elements sometimes require a large image, in which case the ImageLarge property will be used instead.

ImageTransparentColor
A collection that contains the top level set of context menu items.

Text
This is the main text used by attached controls for display.

TextLine1 TextLine2
These two text values are for use by *KryptonRibbon* elements that require two display strings. When a *KryptonRibbon* button is displayed inside a group in large format it draws the two lines of text as provided by these properties.

Checked CheckState
Some attached controls can provide checked state feedback. So if you attached a *KryptonCheckButton* to this command it will be able to display correctly the *Checked* property value which is a boolean type. But it will not display the *CheckState* value because the *KryptonCheckButton* has no way to indicate an indeterminate state. If you attach a *KryptonCheckBox* then the CheckState property would be used. If the attached control cannot provide checked information, such as a *KryptonButton*, then these properties would be ignored by that attached control.

Enabled
Determines if the command should be enabled or disabled for use. When disabled, any attached controls will show themselves as disabled in order to prevent the command from being executed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

KryptonContextMenu

Use the *KryptonContextMenu* component instead of the *ContextMenuStrip* for displaying context menus to the end user. The *Krypton* implementation has several advantages over the *ContextMenuStrip* version. It has additional functionality available on the menu entries, such as the ability to have split buttons. It has additional types of menu entry, such as the header menu item. It also allows for more complex column arrangements.

Implementation
Note that the *KryptonContextMenu* is not just a customized version of the *ContextMenuStrip*. It is written as an entirely new component so you will not be able to convert to using the **Krypton** version by just renaming your existing context menu instances. Nor can you add *ToolStrip* menu instances to the *Krypton* implementation.

KryptonContextMenu Properties

Figure 1 shows the list of properties exposed by the *KryptonContextMenu* component.

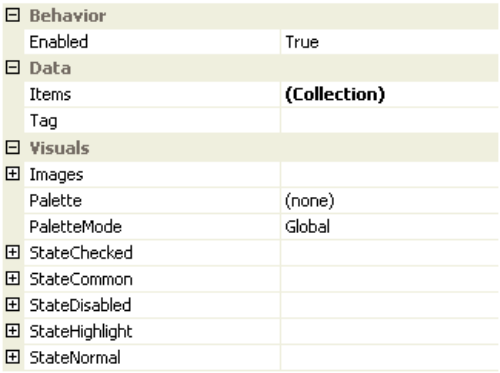


Figure 1 - KryptonContextMenu properties

Enabled
This property does not prevent the context menu from being displayed but it does force all the menu items to be disabled even thought the items themselves are not individually disabled. This *Enabled* state is used to determine if the displayed context menu uses the enabled or disabled border and background for the entire context menu area.

Items
A collection that contains the top level set of context menu items.

Tag
Use the *Tag* to assign your application specific information with the component instance.

Images
Most aspects of the appearance are specified via the set of state properties but there are some elements of the context menu that use images. This compound properties allows you to override the default images that are inherited from the associated palette. Examples of the images it provides are the *Checked* and *Indeterminate* pictures used in the image column for a menu item that has a checked state defined.

Palette
PaletteMode
By default your component will use the globally defined palette. If you need to alter this so that a fixed builtin palette is specified then update the *PaleteMode* property directly. If you need to specify a *KryptonPalette* instance then assign it to the *Palette* property.

StateChecked, StateCommon, StateDisabled, StateHighlight & StateNormal
In order to customize the appearance you need to update the appropriate state properties. Not all elements of the appearance use all of the different possible states, so examine all the different states to discover which are appropriate for the element of interest. The intention of the context menu is that the appearance does not change whilst it is already displayed, so attempting to alter state properties will not update immediately as you might be expecting.

KryptonContextMenu Methods

There are only two methods that you need to be aware of in order to use the *KryptonContextMenu* effectively.

Show

Use the *Show*method to display the context menu. There are several different overrides for the method that allow various ways of specifying the screen location of the menu.

Close
Call *Close* to remove the context menu from display. This method has no effect if the menu is not currently being displayed.

KryptonContextMenu Events

There are four evens exposed by the *KryptonContextMenu* component as seen in figure 3.

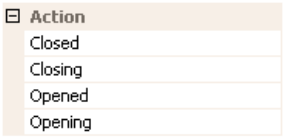


Figure 3 - KryptonContextMenu events

Closed
Generated when the context menu has been removed from display. This event is useful when you need to perform cleanup once the context menu has been dismissed. Note that the event provides information about the reason for the close occurring, so you can take alternative action depending on the cause the menu close.

Closing
Occurs when the context menu is requesting that it be removed from the display. This can occur because the user has clicked on one of the menu options. You can prevent the close from occuring by setting the *Cancel* property of the event arguments to *False*. Event data will contain an enumeration that indicates the reason for the close occuring. So you can decide if the close should be allowed depending on the reason. Note that it is possible for the context menu to be removed without the generation of the *Closing* event. In that case you will receive only the *Closed* event.

Opened
Fired when the context menu has been displayed.

Opening
Generated when the context menu is about to be displayed but before the *Items* collection has been processed. This event has a *Cancel* property so you can prevent the menu from being displayed. This is the appropriate event to use for updating the *Items* collection of entries so that it reflects accurately your application state. This prevents the need to constantly update menu item state during normal operation when the context menu is not being shown.

Menu Items

There are several different types that can be used inside the context menu *Items* hierarchy. However, not all of them are valid at all locations in the hierarchy. At the top level you cannot place an individual *KryptonContextMenuItem* but instead must place them inside of an *KryptonContextMenuItems* collection instead. This extra hierarchy level allows for greater display flexibility because the collection of items can be modified as a group. For example, you can specify that a group of items be displayed without the image column having the traditional background displayed.

The list of items that can be placed at the root *Items* level is as follows...

KryptonContextMenuCheckBox
KryptonContextMenuCheckButton
KryptonContextMenuColorColumns
KryptonContextMenuHeader
KryptonContextMenuImageSelect
KryptonContextMenuItems
KryptonContextMenuItem
KryptonContextMenuLinkLabel
KryptonContextMenuMonthCalendar
KryptonContextMenuRadioButton
KryptonContextMenuSeparator

The list of items that can be placed inside a *KryptonContextMenuItems* is as follows...

KryptonContextMenuItem
KryptonContextMenuHeader
KryptonContextMenuSeparator

KryptonContextMenuCheckBox

Figure 1 shows the list of properties exposed by the KryptonContextMenuCheckBox component.

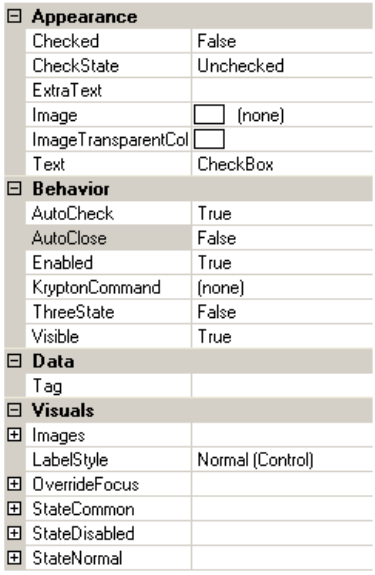


Figure 1 - KryptonContextMenuCheckBox properties

Checked
A boolean property that determines if the check box should be displayed as checked.

CheckState
This property has is useful when you need a three state check box. You can define the property as either *Checked*, *Unchecked* or *Indeterminate* with the *Checked* and *Indeterminate* both causing the *Checked* property to be defined as *True*.

ExtraText
Optional extra text that can appear in addition to the main *Text* property.

Image
Optional image to draw with the text and extra text.

ImageTransparentColor
If you provide an *Image* then this property is used to specify which image color to consider transparent.

Text
This is the standard text that appears as the check box description.

AutoCheck
Should clicking the item cause the checked state to be toggled.

AutoClose
Determines if pressing this check box should cause the context menu to be closed.

Enabled
Should the check box be displayed as enabled and allow interaction with the user. Note that if the *KryptonContextMenu* component has its own *Enabled* property defined as *False* then the item will be disabled regardless of the individual menu item *Enabled* state.

KryptonCommand
Attached command that is used as a source of state.

ThreeState
Should clicking the box cycle around three states or just the traditional two of *Checked* and *Unchecked*.

Visible
Define this as *False* if you do not want the collection of items to be displayed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

Images
Set of images to use for drawing the check box instead of the default palette entries.

ImageStyle
Drawing style applied to the text, extra text and image content.

OverrideFocus, StateCommon, StateDisabled, StateNormal
Overrides for changing how the item is drawn in various states.

Events

Click
Occurs when the user selects the item.

CheckedChanged
Occurs when the value of the *Checked* property changes, usually because the user clicked the item at runtime.

CheckStateChanged
Occurs when the value of the *CheckState* property changes, usually because the user clicked the item at runtime.

KryptonContextMenuCheckButton

KryptonContextMenuCheckButton

Figure 1 shows the list of properties exposed by the KryptonContextMenuCheckButton component.

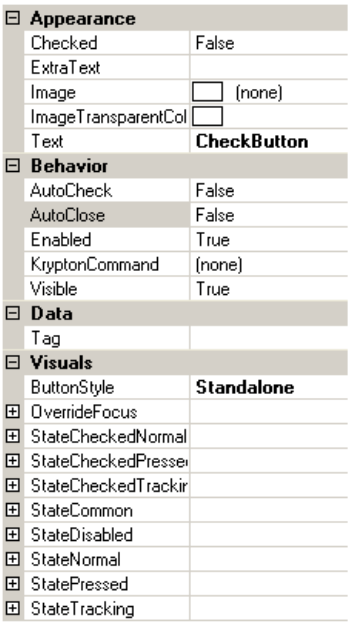


Figure 1 - KryptonContextMenuCheckButton properties

Checked
A boolean property that determines if the check button should be displayed as checked.

ExtraText
Optional extra text that can appear in addition to the main *Text* property.

Image
Optional image to draw with the text and extra text.

ImageTransparentColor
If you provide an *Image* then this property is used to specify which image color to consider transparent.

Text
This is the standard text that appears as the check button description.

AutoCheck
Should clicking the item cause the checked state to be toggled.

AutoClose
Determines if pressing this check button should cause the context menu to be closed.

Enabled
Should the check button be displayed as enabled and allow interaction with the user. Note that if the *KryptonContextMenu* component has its own *Enabled* property defined as *False* then the item will be disabled regardless of the individual menu item *Enabled* state.

KryptonCommand
Attached command that is used as a source of state.

Visible
Define this as *False* if you do not want the collection of items to be displayed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

ButtonStyle
Drawing style applied to the text, extra text and image content.

OverrideFocus, StateCommon etc
Overrides for changing how the item is drawn in various states.

Events

Click
Occurs when the user selects the item.

CheckedChanged
Occurs when the value of the *Checked* property changes, usually because the user clicked the item at runtime.

KryptonContextMenuColorColumns

Figure 1 shows the list of properties exposed by the KryptonContextMenuColorColumns component.

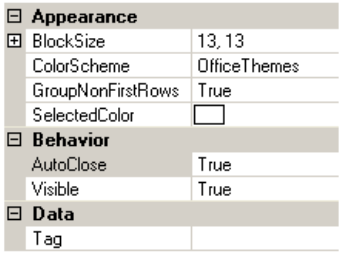


Figure 1 - KryptonContextMenuColorColumns properties

BlockSize
Defines the size of the individual color rectangles.

ColorScheme
Enumeration that specifies the pre-defined set of colors to show.

GroupNonFirstRows
Should the second and subsequent rows of colors be displayed vertically adjacent to each other.

SelectedColor
The color that should be displayed as the initial selection, if that color is defined.

AutoClose
Determines if selecting a new color should cause the context menu to be closed.

Visible
Define this as *False* if you do not want the collection of items to be displayed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

Events

SelectedColorChanged
Occurs when the value of the *SelectedColor* property changes, usually because the user clicked a color block at runtime.

TrackingColor
Generated as the user tracks over different colors, can be used to provide instant feedback to the user about the changes that would occur if that tracking item were to be selected by the user.

KryptonContextMenuHeader

Figure 1 shows the list of properties exposed by the *KryptonContextMenuHeader* component.

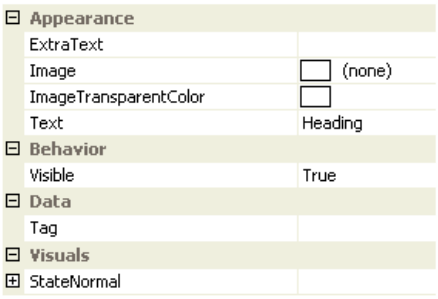


Figure 1 - KryptonContextMenuHeader properties

ExtraText
Optional extra text that can appear in addition to the main *Text* property.

Image
Optional image to draw next to the heading text.

ImageTransparentColor
If you provide an *Image* then this property is used to specify which image color to consider transparent.

Text
This is the standard text that appears as the heading.

Visible
Define this as *False* if you do not want the heading to be displayed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

StateNormal
Properties that allow the customization of the heading appearance when displayed.

KryptonContextMenuImageSelect

Figure 1 shows the list of properties exposed by the KryptonContextMenuImageSelect component.

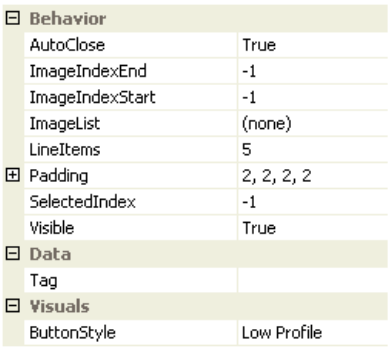


Figure 1 - KryptonContextMenuImageSelect properties

AutoClose
Determines if selecting a new image should cause the context menu to be closed.

ImageIndexEnd
ImageIndexStart
Defines the start and end indexes to use from the *ImageList* for display.

ImageList
Source image list instance that provides the images for display.

LineItems
Number of images to display per horizontal line.

Padding
Pixel spacing to provide as an indent before the contained images are shown.

SelectedIndex
Index into the image list that is the currently selected item.

Visible
Define this as *False* if you do not want the item to be displayed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

ButtonStyle
Style used to draw each of the display images.

Events

SelectedIndexChanged
Occurs when the value of the *SelectedIndex* property changes, usually because the user clicked an image at runtime.

TrackingImage
Generated as the user tracks over different images, can be used to provide instant feedback to the user about the changes that would occur if that tracking item were to be selected by the user.

KryptonContextMenuItems

Figure 1 shows the list of properties exposed by the KryptonContextMenuItems component.

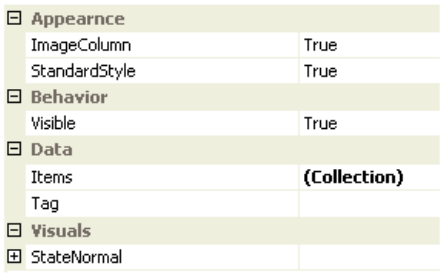


Figure 1 - KryptonContextMenuItem properties

ImageColumn
Should the background of the image column area be drawn.

StandardStyle
Menu items inside this collection can be drawn with one of two styles. If this property is defined as *False* then the alternative style is used. This alternative style will show the menu item *Text* in bold and show the *ExtraText* below the main *Text*. This property is usually modified along with the *ImageColumn* property.

Visible
Define this as *False* if you do not want the collection of items to be displayed.

Items
Collection of child items that are defined to appear in this position.

Tag
Use the *Tag* to assign your application specific information with the component instance.

StateNormal
Properties that allow the customization of the items area when displayed, this applies to the background of the items area and the image column if it is displayed.

KryptonContextMenuItem

Figure 1 shows the list of properties exposed by the KryptonContextMenuItem component.

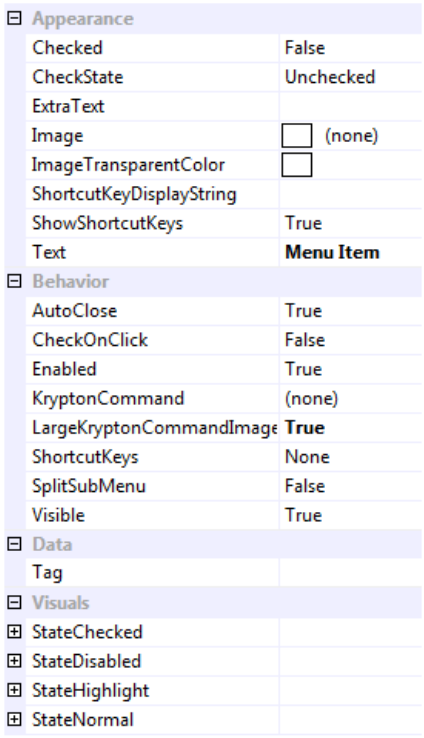


Figure 1 - KryptonContextMenuItem properties

Checked
A boolean property that determines if the menu item should be displayed as a checked item.

CheckState
This property has is useful when you need a three state check item. You can define the property as either *Checked*, *Unchecked* or *Indeterminate* with the *Checked* and *Indeterminate* both causing the *Checked* property to be defined as *True*.

ExtraText
Optional extra text that can appear in addition to the main *Text* property.

Image
Optional image to draw next to the menu item text.

ImageTransparentColor
If you provide an *Image* then this property is used to specify which image color to consider transparent.

ShortcutKeyDisplayString
When a *ShortcutKeys* property value is provided the short cut key combination is shown by default to the right hand side of the menu item. However you can use this property to override that string and force your own display string to be shown instead.

ShowShortcutKeys
When a *ShortcutKeys* property value is provided the short cut key combination is shown by default to the right hand side of the menu item. Use this boolean property to turn off the display of that key combination or the display of the *ShortcutKeyDisplayString*.

Text
This is the standard text that appears as the menu item description.

AutoClose
Determines if pressing this menu item should cause the context menu to be closed. You might want to set this to *False* if you have an item that when clicked should toggle the *Checked* state without causing the menu to be closed.

CheckOnClick
When defined this will cause the *Checked* state of the item to be toggled whenever the item is clicked.

Enabled
Should the menu item be displayed as enabled and allow interaction with the user. Note that if the *KryptonContextMenu* component has its own *Enabled* property defined as *False* then the item will be disabled regardless of the individual menu item *Enabled* state.

KryptonCommand

Attached command that is used as a source of state.

LargeKryptonCommandImage
By default the menu item will take the small image from the attached *KryptonCommand* instance. If you prefer to use the large image instead then set this property to *True* and it will show that alternate image.

ShortcutKeys
Define the keyboard combination that should invoke this item.

SplitSubMenu
If the Items collection is not empty then defining this property will cause the item to be split into two distinct areas. The left area will act as a traditional menu item that when clicked will dismiss the context menu and generate a click event. The right area contains a sub menu indicator and when clicked causes the sub menu to be shown.

Visible
Define this as *False* if you do not want the menu item to be displayed.

Items
Collection of child items that when not empty causes a sub menu indicator to be shown on the right of the menu item. When hovering over the item or when clicked this would also cause the sub menu to be displayed. If you require the menu item to still generate a click event when it has a sub menu then you should consider using the *SplitSubMenu* property.

Tag
Use the *Tag* to assign your application specific information with the component instance.

StateDisable, StateNormal, StateChecked & StateHighlight
Properties that allow the customization of the item area when displayed in various different state.

Events

Click
Occurs when the user selects the item.

CheckedChanged
Occurs when the value of the *Checked* property changes, usually because the user clicked the item at runtime.

CheckStateChanged
Occurs when the value of the *CheckState* property changes, usually because the user clicked the item at runtime.

KryptonContextMenuLinkLabel

Figure 1 shows the list of properties exposed by the KryptonContextMenuLinkLabel component.

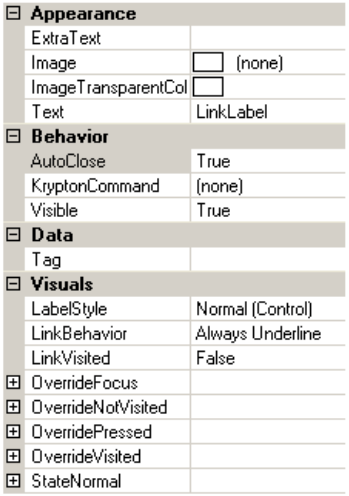


Figure 1 - KryptonContextMenuLinkLabel properties

ExtraText
Optional extra text that can appear in addition to the main *Text* property.

Image
Optional image to draw with the text and extra text.

ImageTransparentColor
If you provide an *Image* then this property is used to specify which image color to consider transparent.

Text
This is the standard text that appears as the link label description.

AutoClose
Determines if pressing this link label should cause the context menu to be closed.

KryptonCommand
Attached command that is used as a source of state.

Visible
Define this as *False* if you do not want the collection of items to be displayed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

LabelStyle
Drawing style applied to the text, extra text and image content.

LabelBehavior
Determine how the link label shows feedback to the user.

LabelVisited
Should the label display as if the link has already been used.

OverrideFocus, StateNormal etc
Overrides for changing how the item is drawn in various states.

Events

Click
Occurs when the user selects the item.

KryptonContextMenuMonthCalendar

Figure 1 shows the list of properties exposed by the KryptonContextMenuMonthCalendar component.

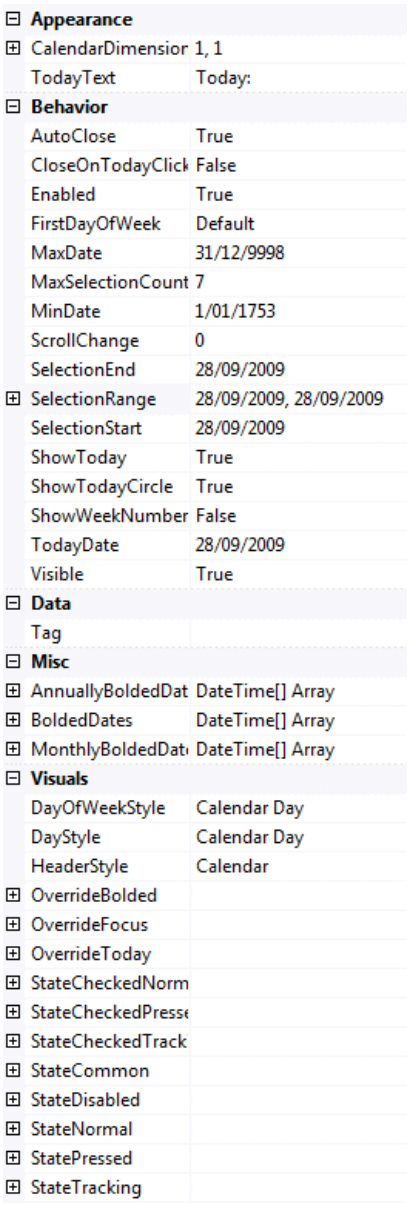


Figure 1 - KryptonContextMenuMonthCalendar properties

CalendarDimensions
Determines the number of months shown as a grid.

TodayText
Text used to caption the today's date in the bottom caption area.

AutoClose
Determines if pressing a day should cause the context menu to be closed.

CloseOnTodayClicked
Determines if pressing the today button should cause the context menu to be closed.

Enabled
Should the month calendar be displayed as enabled and allow interaction with the user. Note that if the *KryptonContextMenu* component has its own *Enabled* property defined as *False* then the item will be disabled regardless of the individual menu item *Enabled* state.

FirstDayOfWeek
Specify which day of week is used as the first displayed column within the month.

MaxDate, MinDate
Place limits on the displayed and selectable date range by using these two properties.

MaxSelectionCount
Number of days that can be selected as a range. Use a value of 1 to allow only a single day to be selected.

ScrollChange
How many months to move forward and backward when the user clicks the next/previous buttons in the top caption.

SelectionRange, SelectionStart, SelectionEnd
Set these properties to define the current selection range and get these values to find the new range when the context menu has been dismissed.

ShowToday, ShowTodayCircle
Determines if today's date is displayed in the bottom caption of the control and if the day that represents today's date has a highlighted border (circle).

ShowWeekNumbers
When defined this property will show week numbers in the row header of each displayed week of values.

TodayDate
Defaults to the current date when the control is created but can be set to define any date as the today date.

Visible
Define this as *False* if you do not want the element to be displayed.
Tag
Use the *Tag* to assign your application specific information with the component instance.
AnnuallyBoldedDates, MonthlyBoldedDates, BoldedDates
Use these collections to specify which dates should be displayed in a bold state.
DayOfWeekStyle, DayStyle, HeaderStyle
Properties used to define the palette styles used to draw various elements of the control.
OverrideFocus, OverrideBolded, OverrideToday StateCommon, StateDisabled, StateNormal, StateTracking, StatePressed StateCheckedNormal, StateCheckedTracking, StateCheckedPressed
Overrides for changing how the month elements are drawn in various states.

Events

DateChanged
Occurs when any of the date properties is changed.
SelectionStartChanged
Occurs when the *SelectionStart* property changes.
SelectionEndChanged
Occurs when the *SelectionEnd* property changes.

KryptonContextMenuRadioButton

Figure 1 shows the list of properties exposed by the KryptonContextMenuRadioButton component.

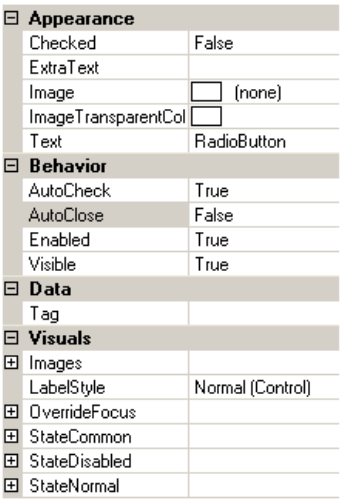


Figure 1 - KryptonContextMenuRadioButton properties

Checked
A boolean property that determines if the radio button should be displayed as checked.

ExtraText
Optional extra text that can appear in addition to the main *Text* property.

Image
Optional image to draw with the text and extra text.

ImageTransparentColor
If you provide an *Image* then this property is used to specify which image color to consider transparent.

Text
This is the standard text that appears as the radio button description.

AutoCheck
Should clicking the item cause the checked state to be toggled.

AutoClose
Determines if pressing this radio button should cause the context menu to be closed.

Enabled
Should the check button be displayed as enabled and allow interaction with the user. Note that if the *KryptonContextMenu* component has its own *Enabled* property defined as *False* then the item will be disabled regardless of the individual menu item *Enabled* state.

Visible
Define this as *False* if you do not want the collection of items to be displayed.

Tag
Use the *Tag* to assign your application specific information with the component instance.

Images
Set of images to use for drawing the radio box instead of the default palette entries.

ImageStyle
Drawing style applied to the text, extra text and image content.

OverrideFocus, StateCommon, StateDisabled, StateNormal
Overrides for changing how the item is drawn in various states.

Events

Click
Occurs when the user selects the item.

CheckedChanged
Occurs when the value of the *Checked* property changes, usually because the user clicked the item at runtime.

KryptonContextMenuSeparator

KryptonContextMenuSeparator

Figure 1 shows the list of properties exposed by the KryptonContextMenuSeparator component.

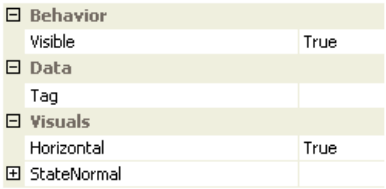


Figure 1 - KryptonContextMenuSeparator properties

Visible

Define this as *False* if you do not want the separator to be displayed.

Tag

Use the *Tag* to assign your application specific information with the component instance.

Horizontal

The separator can be used to provide either a vertical or horizontal break in the display. When defined as *True* it will cause a new column to be started for items that follow the separator in the parent collection and draw a vertical separator between the two columns.

StateNormal

Properties that allow the customization of the separator appearance when displayed.

KryptonPalette

Use the *KryptonPalette* component to define or modify a global palette. You can then assign use of the palette to individual *Krypton* controls. Alternatively you can update the *KryptonManager* to use it as the global palette and so all *Krypton* controls will be affected that have not already been given a custom palette to use.

BasePaletteMode
BasePaletteMode is used to determine how to inherit individual palette values that have not been specified in the palette component itself. When you create a new *KryptonPalette* instance all the palette values are defined to inherit from the base palette.

If you choose *Global* as the base palette then values will inherit from whatever has been specified as the global palette in the *KryptonManager* component. Other values include built-in palettes such as the *Professional - Office 2003* and the *Professional - System* options. The last option is called *Custom* and is valid only when you provide a palette reference to the *BasePalette* property.

BasePalette
BasePalette should be used when you need to inherit from another *KryptonPalette* instance instead of the global palette or one of the built-in palettes. Once you assign a reference to this property the *BasePaletteMode* will automatically be changed to the *Custom* enumeration value.

Note that you are not allowed circular references in the use of base palettes. So if you define the base to be another palette instance and then try to make the other palette instance references back to this one then it will cause an error. Whenever you alter the base palette it will check the chain to ensure no circular dependency exists.

Use of the base palette is best explained with a couple of example scenarios. Imagine you are using the *Professional - System* built-in palette for your *Krypton* controls. You can set this up by altering the global palette to be *Professional - System* on the *KryptonManager* component. Now you discover that you need to alter just one aspect of the appearance such as the border control of all the button controls.

To solve this you need to create a *KrytonPalette* instance and define the base palette as the *Professional - System* built-in palette. Then you update the border color setting for the button control to the required value, all other values will inherit from the base palette and so this is the only custom value that needs defining. Finally alter the *KryptonManager* so it uses our new *KryptonPalette* as the global palette.

A variation on this scenario is where you need to provide this custom border to a group of *Krypton* controls but not to all of them. In this case you would create the new *KrytonPalette* as before but this time you would inherit it from the *Global* palette. Then you assign use of the new palette to each of the individual controls that need the altered appearance. All the other controls will continue to use the *Global* palette and so be unaffected.

BaseRendererMode & BaseRenderer
Each builtin palette specifies the renderer instance to use when drawing. The *Office 2007* builtin palettes use the *Office2007* renderer class and the set of *Professional* builtin palettes use the associated *Professional* renderer class. Different render implementations are used for different palettes to provide custom drawing actions that are palette specific.
In the unlikely event you need to alter the renderer used you can modify the *BaseRendererMode* property. If you create your own renderer or a renderer that derives from a builtin one then you would assign it to the *BaseRenderer* property. By default the *BaseRendererMode* property is defined as *Inherit* and so the palette inherits the render setting from the base palette.
AllowFormChrome
Some palettes require that *KryptonForm* instances perform custom chrome drawing in the border and caption areas. The default value of *Inherit* specifies that the setting by retrieved from the base palette setting. For example the *Office 2007 - Blue* builtin palette will provide *True* for this property but the *Professional - Office 2003* will return *False*. To turn off custom chrome then you would set this property to *False*.

Palette Values
The set of properties for each type of style follow the same basic structure. At the top level is the name of the a styles collection such as *ButtonStyles*, *LabelStyles*, *PanelStyles* etc. Within this collection are contained each of the individual style instances. For example the *PanelStyles* property contains *PanelClient*, *PanelAlternate*, *PanelCustom1* and *PanelCommon*. The first three of these are *PanelClient*, *PanelAlternate* and *PanelCustom1* which are actual styles you can choose when using a *KryptonPanel* instance. *PanelCommon* is the base style that the other three inherit from, so you can place common settings in *PanelCommon* and know that the other three styles will inherit those settings.

For each individual style there are child properties for each of the possible states associated with that style. So the *PanelClient* style contains *StateDisabled*, *StateNormal* and *StateCommon*. Here the *StateDisabled* and *StateNormal* relate to the actual states of the *PanelClient* style and the *StateCommon* is the base set of values that the *StateDisabled* and *StateNormal* inherit from.
If you examine the *KryptonPanel* control that uses the *PanelClient* style you will see it also has the same three set of state values that can be overridden on a per control basis. This hierarchy of style collection, individual style and style states can be seen in figure 1.

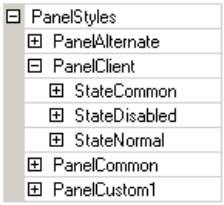


Figure 1 - Panel styles hierarchy

If you do not set a value for any of the style states such as *StateNormal*, *StateDisabled* and *StateCommon* then it inherits the values by using a top level group called *Common*. There are three entries in the *Common* collection called *StateDisabled*, *StateOthers* and *StateCommon*. The style specific *StateDisabled* will inherit from the palette level *StateDisabled* if the style specific *StateCommon* is not defined. All other style specific states inherit from *StateOthers* if the style specific *StateCommon* is not defined. Both of the palette level *StateDisabled* and *StateOthers* inherit from the *StateCommon* values. Only if the palette level *StateCommon* value is not defined will it use the base palette.

This is illustrated with a simple example. Imagine we have a *KryptonPanel* control instance with a style of *PanelClient* that is using our *KryptonPalette* instance as the defined palette. The control is in the normal state and so uses the following inheritance sequence to discover display values to use: -

(Look for a value at the individual control level)
KryptonPanel.StateNormal
KryptonPanel.StateCommon
(Look for a value at the palette level)
KryptonPalette.PanelStyle.PanelClient.StateNormal
KryptonPalette.PanelStyle.PanelClient.StateCommon
KryptonPalette.Common.StateOthers
KryptonPalette.Common.StateCommon

The advantage of this inheritance chain is the speed with which you can customize at the appropriate level. If you want to set the background color of all *Krypton* controls then you would alter the *KryptonPalette.Common.StateCommon* level values. To alter the background for just all the panel styles then you would change *KryptonPalette.PanelStyles.PanelCommon.StateCommon* values. To change the background of just the PanelClient you would alter *KryptonPalette.PanelStyles.PanelClient.StateCommon*.

ToolMenuStatus
You can use the palette to alter not just the styling of the *Krypton* controls but also the appearance of the tool strips, menu strips, status strips and context menu strips. This is the purpose of the *ToolMenuStatus* property on the palette.

Below this property are a whole range of property collections that related to a particular area such as *ToolStrip* and *StatusStrip*. You can navigate and alter values as appropriate to achieve the customized look you need that will match the rest of the palette settings.

A good way to work with these settings is to alter the *KryptonManager* so the palette is the global palette. Now any change you make will be instantly reflected in the tool, menu and status strips on your form. Then try changing the color of interest to be red in order to see the effect it has on the display. This will then point out how to them modify the color to the actual value you require.

Import and Export
At design time you can use the smart tag of the *KryptonPalette* to invoke the exporting or importing of palette settings. Exporting will allow you to generate an XML file that contains all the values that have been changed from the default. You can then import these settings into another *KryptonPalette* instance in the same or a different application. Use the *Reset* option from the smart tag to reset all the palette properties back to the default settings that have when the component is first created.

KryptonInputBox

The *KryptonInputBox* provides access to a *Krypton* style dialog box that allows the user to enter a string. It is a replica of the *InputBox* control that is provided as part of the *Microsoft.VisualBasic* assembly and is typically used by *VB.NET* developers. By providing a Krypton version of this dialog it ensures that your whole application has a consistent look and feel that extends to even the input boxes that appear.

Appearance

The displayed *KryptonInputBox* derives from the *KryptonForm* base class and so has the same appearance as other *Krypton* style forms. In order to show the *Krypton* input box you need to call one of the static methods it exposes called *Show*. The simplest method takes a single parameter and will show an input box with the specified prompt text. All the other parameters will be defaulted. Use one of the other methods if you need greater control window title or default input string. See Figure 1 for an example of the input box in operation.

String Localization

The button text will always display in English by default. If you need to localize the strings to other languages you can do so by placing a *KryptonManager* component on your main *Form*. Use the properties window and then expand the *GlobalStrings* property and modify the strings as needed.

Figure 1 – Example Appearance

KryptonManager

Use the *KryptonManager* component to modify global settings that affect all the *Krypton* controls in your application. Note that the global settings affect all controls and not just those on the same form as the *KryptonManager* instance.

Global Palette
Each individual *Krypton* control will by default inherit its display values from the global palette that is specified by the *KryptonManager*. This makes it easy to change the global palette and have all *Krypton* controls update their appearance in one step.

GlobalPaletteMode is an enumeration that specifies which palette to use as the global palette. You can either choose one of the built-in palettes such as the *Professional - Office 2003* or specify use of a custom palette.

GlobalPalette should be used when you need to use a *KryptonPalette* instance instead of one of the built-in palettes. Once you assign a reference to this property the *GlobalPaletteMode* will automatically be changed to the *Custom* enumeration value.

Note that you are not allowed circular references in the use of palettes. So if you define the global palette to be a *KryptonPalette* and then try to make the base palette of the *KryptonPalette* the *Global* value then it will cause an error. Whenever you alter the global palette or the base palette of a *KryptonPalette* it will check the inheritance chain to ensure no circular dependency is created.

GlobalAllowFormChrome
When your *Form* derives from the *KryptonForm* base class the caption and border areas will be custom painted if the associated *Form* palette indicates it would like custom chrome. For example the *Office 2007* builtin palettes all request that the form be custom drawn to achieve a consistent look and feel to that of *Microsoft Office 2007* applications. If you would like to prevent any of your *KryptonForm* derived windows from having custom chrome then set this property to *False*. This setting overrides any requirement by a palette or *KryptonForm* to have custom chrome.

GlobalApplyToolstrips
In order to ensure your entire application looks consistent the *KryptonManager* will create and assign a tool strip renderer to your application. Whenever you change the global palette or the palette has a tool strip related value changed the tool strip renderer is updated to reflect this. If you prefer to turn off this feature so that you control the tool strip rendering manually then you just need to assign *False* to the *GlobalApplyToolstrips* property.

GlobalStrings
The *KryptonMessageBox* has buttons that use the string values from this section. If you would like to alter the strings displayed on those buttons then you can do so by altering the values in this area. If you have set the *Localization* property on your *Form* to *True* then these values will be stored on a per-language setting allowing your message box to have different display strings per language you choose to define.

KryptonMessageBox

The *KryptonMessageBox* provides access to a *Krypton* style message box that displays a modal dialog with text, buttons and symbols that inform and instruct the user. This ensures that your whole application has a consistent look and feel that extends to even the message boxes that appear.

Appearance

The displayed *KryptonMessageBox* derives from the *KryptonForm* base class and so has the same appearance as other *Krypton* style forms. In order to show the *Krypton* message box you need to call one of the many static methods it exposes called *Show*. The simplest method takes a single parameter and will show a message box with the specified message content. All the other parameters will be defaulted. Use one of the other methods if you need greater control over the buttons, symbols etc. See Figure 1 for an example of the message box in operation.

String Localization

The button text will always display in English by default. If you need to localize the strings to other languages you can do so by placing a *KryptonManager* component on your main *Form*. Use the properties window and then expand the *GlobalStrings* property and modify the strings as needed.

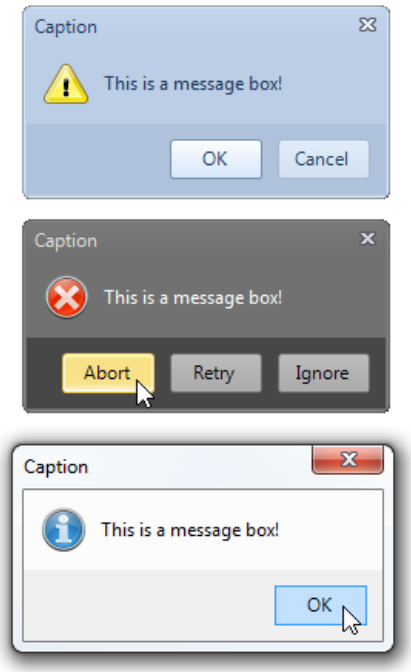


Figure 1 – Example Appearance

KryptonTaskDialog

The *KryptonTaskDialog* provides an extended version of the standard *MessageBox* but with more flexibility. The windows *TaskDialog* was introduced with *Windows Vista* and the Krypton version is intended to provide most, but not all, of the same functionality. Try using the KryptonTaskDialogExamples sample that comes with the *Toolkit* in order to see and experiment with the *KryptonTaskDialog* implementation. Figure 1 shows an example of the KryptonTaskDialog with all the possible options used.

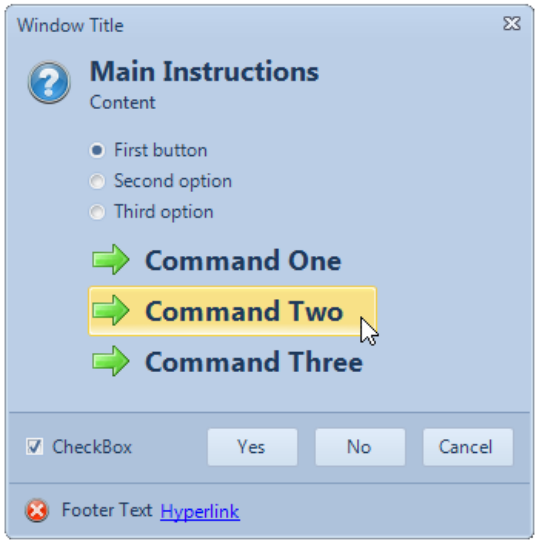


Figure 1 – KryptonTaskDialog Example

Usage

You can invoke the *KryptonTaskDialog* in one of two ways. The quickest and easiest is to use one of the overrides for the static *Show* method. This can be called without the need to create an instance of any component and will return a *DialogResult* as the result of the operation. This method of operation mimics the static *Show*methods that exist for the *KryptonMessageBox* and standard .NET *MessageBox* components. Here is an example code showing how this can be achieved:-

```
DialogResult result = KryptonTaskDialog.Show("Window Title",
                                             "Main Instructions",
                                             "Content",
                                             MessageBoxIcon.Information,
                                             TaskDialogButtons.OK |
                                             TaskDialogButtons.Cancel);


switch (result)     {
    case DialogResult.OK:
        break;

    case DialogResult.Cancel:
        break;
}
```

A limitation with the above approach is that not all the functionality of the task dialog can be accessed in this manner. To enable use of all the possible features you should use the second approach. This involves creating an instance of the *KryptonTaskDialog* and then setting properties on the instance before calling the *ShowDialog* method of the component. On return from the *ShowDialog* call you can then examine the properties of the component that have been updated by the user operating the dialog. For example, the *KryptonTaskDialog.CheckboxState* will be updated with the checked state of the check box control that was shown to the user in the dialog. Here is an example of using the second approach:-

```
using(KryptonTaskDialog kryptonTaskDialog = new KryptonTaskDialog())     {
    kryptonTaskDialog.WindowTitle = "Window Title";
    kryptonTaskDialog.MainInstruction = "Main Instruction";
    kryptonTaskDialog.Content = "Content";
    kryptonTaskDialog.Icon = MessageBoxIcon.Warning;
    kryptonTaskDialog.CommonButtons = TaskDialogButtons.OK | TaskDialogButtons.Cancel;
    kryptonTaskDialog.DefaultButton = TaskDialogButtons.OK;
    kryptonTaskDialog.FooterText = "Footer Text";
```

```
        kryptonTaskDialog.FooterHyperlink = "Hyperlink";
        kryptonTaskDialog.FooterIcon = MessageBoxIcon.Error;
        kryptonTaskDialog.CheckboxText = "Checkbox Text";
        kryptonTaskDialog.CheckboxState = false;
        kryptonTaskDialog.AllowDialogClose = true;
        DialogResult result = kryptonTaskDialog.ShowDialog();

        switch (result)         {
            case DialogResult.OK:
                break;

            case DialogResult.Cancel:
                break;          }
    }
```

To simplify the setting of the properties it would be easier to drag the *KryptonTaskDialog* from the toolbox onto the *Form* and then use the properties window to modify the component properties as needed at design time. Then you need only invoke the *ShowDialog* method and process the result rather than manually writing the code to set the properties. This also makes it easier to hook into the events generated by the component.

String Localization

The dialog buttons text will always display in English by default. If you need to localize the strings to other languages you can do so by placing a *KryptonManager* component on your main *Form*. Use the properties window and then expand the *GlobalStrings* property and modify the strings as needed.

KryptonTaskDialog Properties

Figure 2 shows a list of all the properties that can be used to define the appearance of the component.

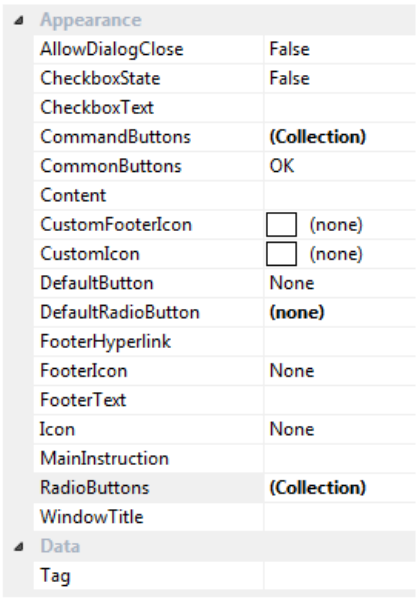


Figure 2 – KryptonTaskDialog Properties

AllowDialogClose

The dialog will only have a window close button if either this property is defined as *True* or the *Cancel* button is one of the *CommonButtons*, otherwise the dialog will not have a close button. When the dialog is allowed to be closed the user can use the *ESCAPE* key, the *ALT+F4* key combination or the window close button.

CheckboxState
CheckboxText

If the *CheckboxText* is defined with a string then a check box control will appear on the dialog buttons area of the window with an initial state of *CheckboxState*. When the window has been dismissed the *CheckboxState* will be updated with whatever value the user defined whilst it was showing.

CommonButtons
DefaultButton

The set of dialog buttons displayed is determined by the *CommonButtons* setting. *DefaultButton* specifies which of the showing dialog buttons should be the default and so initially focused when the dialog is displayed.

RadioButtons
DefaultRadioButton
CommandButtons
Just below the main text of the dialog a set of radio buttons are displayed in response to the entries in the *RadioButtons* collection. The initial radio button to be checked is defined by the *DefaultRadioButton* property. This property is updated as the user selects different radio buttons at runtime and so once the dialog is dismissed you can examine the DefaultRadioButton property to find out what entry the user selected. The *CommandButtons* collection specifies a series of display buttons below the radio buttons. If the user selects one of these buttons then the *DialogResult* defined for that *CommonButtons* entry is returned as the result of showing the dialog.

FooterText FooterHyperlink
FooterIcon
CustomFooterIcon
The footer area will be shown if any of the footer related properties has been defined. The *CustomFooterIcon* is used as the source of the footer icon if defined, otherwise the *FooterIcon* property is used instead. To the right of the icon any *FooterText* string is shown and finally the *FooterHyperlink* string is shown as a link label that can be clicked. If the user clicks the link label then the *KryptonTaskDialog.FooterHyperlinkClicked* event is fired so you can perform an appropriate action.

WindowTitle MainInstruction
Content
Icon
CustomIcon
These properties represent the basic settings of the dialog. The *WindowTitle* is used as the caption shown in the dialog caption bar. *MainInstruction* and *Content* are strings displayed in the main section of the dialog client area. The *CustomIcon* is used as the source of the main icon if defined, otherwise the *Icon* property is used instead. See Figure 1 for an example of how the various properties are displayed at runtime.

Tag
Custom data field for use by the application developer.

KryptonTaskDialog Events
There is just a single event of interest. The *KryptonTaskDialog.FooterHyperlinkClicked* event is fired when the user clicks the footer hyper link label. This occurs whilst the dialog is still being shown.

KryptonForm

The *KryptonForm* derives from *Form* and is intended as the base class for all your application *Form* instances. So instead of inheriting your window classes from *Form* you should always inherit from *KryptonForm*. The purpose of the *KryptonForm* is to perform custom chrome painting so that the border and caption areas of the form are drawn according to the defined palette settings. This ensures that your whole application has a consistent look and feel that extends to the windows themselves as well as the regular *Krypton* controls.

Appearance

The *GroupBackStyle* and *GroupBorderStyle* properties are used to define the border areas of the custom chrome drawing. The default value for both of these properties is *FormMain.* The *HeaderStyle* property is used to define the appearance of the caption area of the custom chrome, it has a default value of *Form*.

Two States

Only two possible states of *Active* and *Inactive* are used by the *KryptonForm* control. In order to customize the appearance use the corresponding *StateActive* and *StateInactive* properties. So if you wanted the custom chrome to have a red border when active but a blue border when inactive you would alter these two states respectively to achieve that customization.
To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border color in *StateActive* and *StateCommon* then the *StateActive* value will be used whenever the control is in the *Active* state. Only if the *StateActive* value is not overridden will it look in *StateCommon*.

AllowFormChrome

The default value for this property is *True* and indicates that the form is allowed to have custom chrome painting if all other criteria are matched. Other criteria include the *KryptonManager.GlobalAllowFormChrome* being set to True as well as the palette being used for drawing having an *AllowFormChrome* setting of *True*. So you will only have custom chrome is the form, palette and global manager allowed it to be used.
If you set this property to *False* then the form will not perform custom painting of borders and caption area no matter what the global manager and palette might request. When custom chrome is not applied then the window shows a standard appearance appropriate for the operating system.

AllowStatusStripMerge

In order to achieve a professional looking appearance it is possible to request that the status strip of the form be merged into the border area. This gives an integrated look can be observed with many of the *Office 2007* applications. To prevent status strip merging just set this property to *False*. When defined as *True*, the default, there are several criteria that must be met before status strip merging will occur.
First of all the form can only merge an instance of the *StatusStrip* control into the border, the older *StatusBar* cannot be merged nor any custom control that you have created. Second the *StatusStrip* must be visible and have a *Dock* value of *DockStyle.Bottom*, thus showing the control is intended to be at the bottom of the form and sized to fill the entire width of the client area. Last of all the StatusStrip must be using a *RenderMode* setting of *ToolStripRenderMode.ManagerRenderMode*. Only if all these criteria are met will the control be considered for merging. Figure 1 shows an example of a status strip merged and not merged so you can see the visual difference.

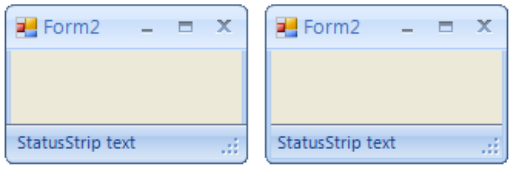


Figure 1 - StatusStripMerging = True & False

AllowButtonSpecTooltips

By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

ButtonSpecs

You can add buttons to the caption area by modifying the *ButtonSpecs* collection exposed by the *KryptonForm*. Each *ButtonSpec* entry in the collection describes a single button for display on the header. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances. See the ButtonSpec section for more details. Figure 2 shows an example of a *KryptonForm* with an extra button.

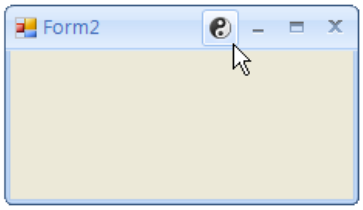


Figure 2 – KryptonForm with extra ButtonSpec

Caption Values

You can specify three values that are then displayed inside the caption area. Two of these are existing properties of the form, *Text* and *Icon*. The third property is a new one called *ExtraText* and allows an additional string to be specified and displayed. Figure 3 shows the result of defining a value for this *ExtraText* property.

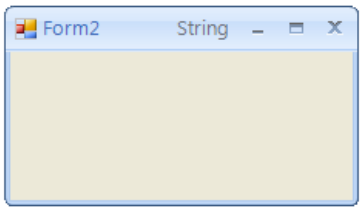


Figure 3 – ExtraText = "String"

Navigator

Learn more about the capabilities of the *KryptonNavigator* using the following sections.

Overview

- Overview
- KryptonPage
- Modes
- Page Dragging

Visual Properties

- PopupPages Properties
- ToolTips Properties
- Other Properties

Mode Operation

- Bar Modes
- Button Modes
- Group Modes
- Header Modes
- Outlook Modes
- Panel Modes
- Stack Modes

Events

- Selection Events
- Action Events
- Other Events

Navigator Overview

The *Navigator* control is designed to provide a variety of ways for the user to navigate around a set of pages. Modes range from the very basic such as *Panel*, which displays no mechanism for the user to switch pages, to the sophisticated such as *Outlook - Full*, where the user can randomly switch to any page using a pleasant user interface experience.

Modes of Operation One of the big advantages this control provides is the use of a single mode property to switch the entire user interface. This simplifies application development because you only have to learn how a single control works, rather than learning about several separate controls where each control provides the equivalent of a single navigator mode. Another advantage is you do not need to remove and add a different control to your form each time you want to alter the user interface. Instead just change the mode and you switch from a traditional TabControl style to a header group style. Figure 1 shows examples of two different modes that are achieved by just altering the mode property.

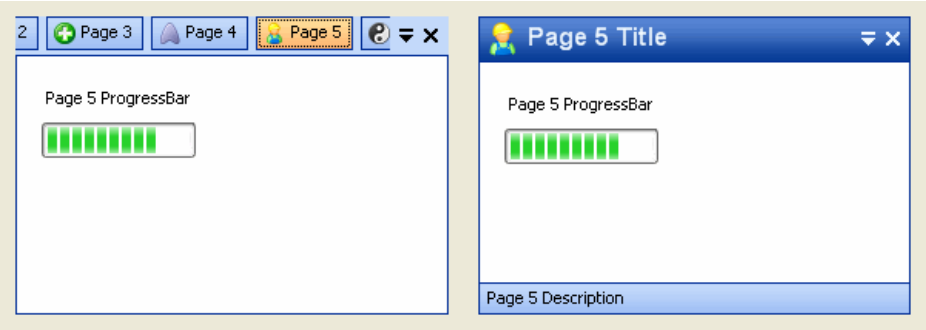


Figure 1 - Two Example Modes

See Navigator Modes for a detailed list of all the different modes available.

Navigator Properties
Properties that control the appearance and operation of the navigator are concentrated in two different categories. Figure 2 shows the set of properties in the *Visuals* category; these are explained here and in the PopupPages Properties and ToolTips Properties sections. Figure 3 shows the *Visuals (Modes)* category which contains mode specific properties that are described in detail in the Bar Modes, Button Modes, Group Modes, Header Modes, Outlook Modes, Panel Modes and Stack Modes sections.

Visuals	
NavigatorMode	Bar - Tab - Group
⊞ OverrideFocus	
PageBackStyle	Control - Client
Pages	**(Collection)**
Palette	(none)
PaletteMode	Global
⊞ PopupPages	
⊞ StateCommon	
⊞ StateDisabled	
⊞ StateNormal	
⊞ StatePressed	
⊞ StateSelected	
⊞ StateTracking	
⊞ ToolTips	

Figure 2 - Visuals

Visuals (Modes)	
⊞ Bar	
⊞ Button	
⊞ Group	
⊞ Header	
⊞ Outlook	
⊞ Panel	
⊞ Stack	

Figure 3 - Visuals (Modes)

Navigator Events
Information about all the different events exposed by the navigator are found in the Selection Events, Action Events and Other Events sections. At a minimum you are recommended to read up on the selection events as they are crucial in understanding how to correctly handle selection changes and provide fine grained control over selection activity.

Pages Property
To add, remove and modify the collection of *KryptonPage* instances associated with the control you just need click the property value

button for the *Pages* property. This will then display the collection editor as can be seen in Figure 4. Note that some changes such as removing pages or changing the ordering of pages will not be reflected immediately in the navigator; not until you press *OK* and leave the collection editor does the control get refreshed.

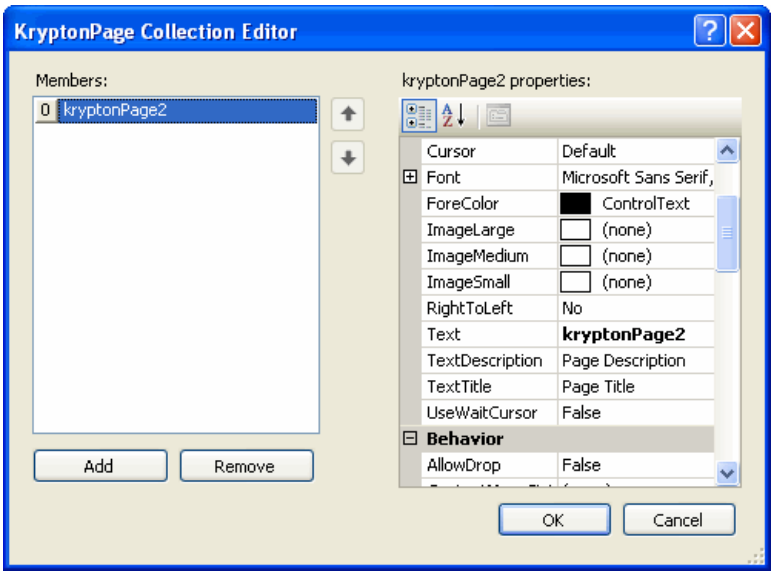


Figure 4 - KryptonPage Collection Editor

PageBackStyle Property

The navigator allows you to use the *PageBackStyle* property to define the appearance of the selected *KryptonPage* background. The default value of *Control - Client* gives an appropriate appearance for most usages but you change the value to any of the background styles. Figure 5 shows the default style on the left and the *Header - Primary* style on the right.

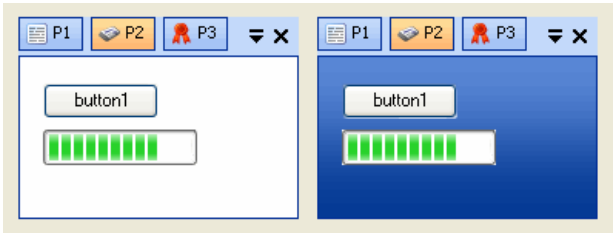


Figure 5 - PageBackStyle Styles

KryptonPage

Each *KryptonPage* has the ability to override the appearance of the navigator when the page is selected and also, to a lesser extent, when not selected. This is useful if you want a particular page to have a different color border or text for its display. Read the KryptonPage section for a detailed description of how to achieve this.

Five States

As with all the *Krypton* controls, each possible state of the control has a set of properties that can be used to customize the appearance. For the navigator there are five states with corresponding property names of *StateDisabled*, *StateNormal*, *StateTracking*, *StatePressed* and *StateSelected*. Unlike most *Krypton* controls the contents of each state are not identical in all cases. This is because different elements that make up the control are themselves capable of achieving different states.

This is best explained with a couple of examples. Many of the navigator modes used a *Group* element for showing the contents of the selected page. A *Group* element is only capable of being in the *Disabled* or *Normal* states and so you will only find properties for customizing the *Group* appearance inside the *StateDisabled* and *StateNormal* properties. But the *CheckButton* items that appear for some of the bar modes make use of all states as they need to represent the three states of *Tracking*, *Pressed* and *Selected* modes.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define a property in both the *StateNormal* and *StateCommon* sets then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Override States

There is an additional state related property called *OverrideFocus* is used to alter the appearance of the control when it has the focus. Notice that the property starts with the *Override* prefix instead of the usual *State*. This is because it does not relate to a specific control state such as *Normal* or *Tracking*. Instead it is applied to any of the other states and is used to override the appearance that would otherwise be shown.

Mnemonics

In order to use mnemonics for the quick selection of different pages you need to add the '&' character into the *KryptonPage.Text* property in front of the character that will act as the mnemonic. This will then be displayed as an underlined character by the navigator. By default the *UseMnemonic* property of the navigator is defined as *True* and so the navigator will search visible pages for a matching mnemonic as you use the keyboard.

Navigator KryptonPage

The *KryptonPage* is the only type that can be added to the *KryptonNavigator.Pages* collection. It works just like a traditional *Panel* in that you can drag and drop controls onto it during design time for display when that page is selected inside the navigator. It does however have extra properties that allow it to work in conjunction with the navigator to achieve the look and feel you need.

Text, TextTitle and TextDescription

You can see in Figure 1 that the *KryptonPage* has three text properties beginning with the word *Text* that are typically used to provide various types of description for the page. The standard *Text* property is intended for giving the shortest description about the page, the kind of brief text appropriate for display on a check button. *TextTitle* is for use when you have more space available and should be used to provide a fuller description of the page. An appropriate use of this property would be a title for a header. The *TextDescription* can contain a very long and detailed description of the page. This would be appropriate for appearing on a status bar.

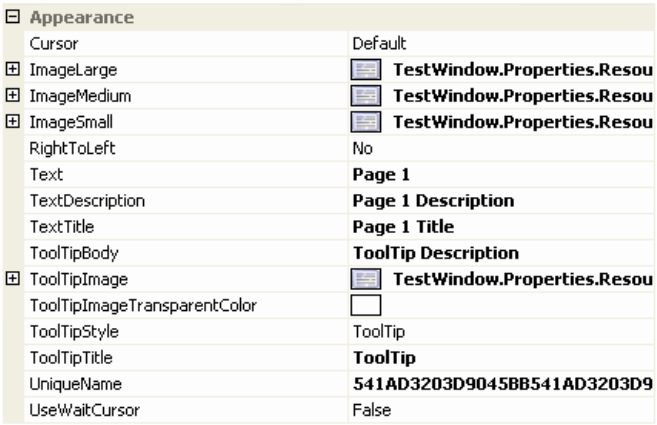


Figure 1 - KryptonPage Appearance Properties

ImageLarge, ImageMedium and ImageSmall

Also in Figure 1 you can see that there are three different image properties beginning with the word *Image*. *ImageSmall* should contain a small image of 16x16 and is appropriate for use inside a check button. Use *ImageMedium* for an intermediate sized image of 24x24 that would be used on a section header. Last is the *ImageLarge* that is recommended to be sized at 48x48 and is for uses as a large icon representation of the page. Note that the 16x16, 24x24 and 48x48 sizes are only recommendations and you can provide any sized image that you like.

ToolTipTitle, ToolTipBody, ToolTipImage, ToolTipImageTransparentColor and ToolTipStyle

These properties are used to specify the tool tip details for display when the user hovers over the page heading. Use the *ToolTipTitle* and *ToolTipBody* properties to define two text strings for display. To associate an image with the tool tip you should assign it to *ToolTipImage* and use the *ToolTipImageTransparentColor* for specifying a color in the image that should be treated as transparent. For example, many bitmaps will use magenta as a color for the background area that should become transparent when the bitmap is drawn, in that case assign *Color.Magenta* to the *ToolTipImageTransparentColor* property.

The default value for the *ToolTipStyle* is *LabelStyle.ToolTip* and will cause the image and title text to be shown at the top of the tool tip area and the body text to be shown below. Alternatively you could change the style to *LabelStyle.SuperTip* in which case the title text is shown in bold at the top of the tool tip area, the image in shown below with the title also below and to the right of the image.

UniqueName Property

As the name suggests, this field allows the developer to assign a unique name to the page instance. By default each new instance will be assigned a new GUID generated from the operating system to ensure uniqueness. As these are not easy to remember it is recommended you alter the property to a more meaningful value if you intend to make use of the property.

Visual Properties

Figure 2 shows the list of properties you can use to override the appearance of the page itself and other navigator elements. Use these properties if you to alter the look and feel on a per-page basis. For example, you might decide that one particular page needs a bright red border to indicate an error condition with some of the controls contained on the page.

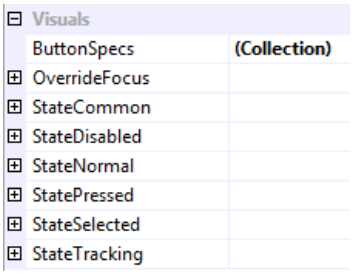


Figure 2 - KryptonPage Visual Properties

Figure 3 shows an example where the appearance of the first page has been altered so that it shows up in distinct red coloring. All the

other pages have been left with default settings. The left image shows the first page selected and the right image when the second page is selected.

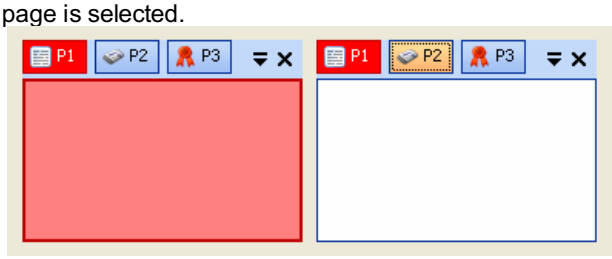


Figure 3 - Customized Page Visuals

Figure 4 shows an example where a button spec has been added to the first page. You and can per-page button spec definitions to add extra buttons into any navigator page.

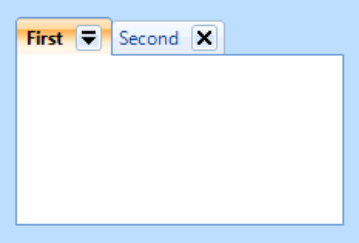
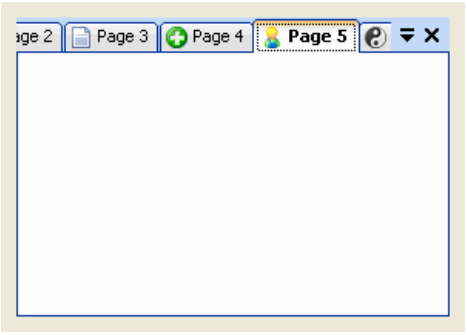


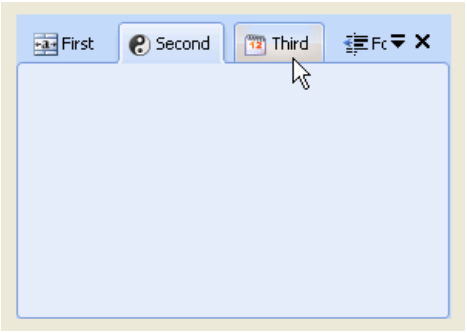
Figure 4 - Pages with ButtonSpec definitions

Navigator Modes

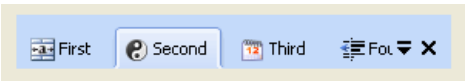


Bar - Tab - Group
A Bar showing Tab headers is placed on the outside of a Group container. Each Tab represents a single Page instance.

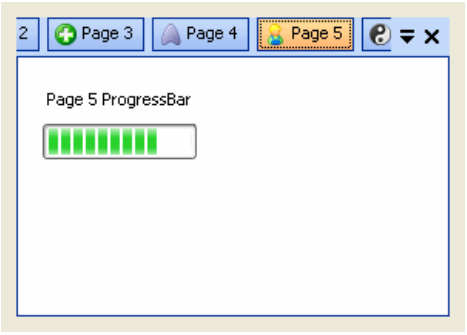
Bar - Tab - Only
This is a tab strip mode showing a Tab header for each Page instance in the navigator.



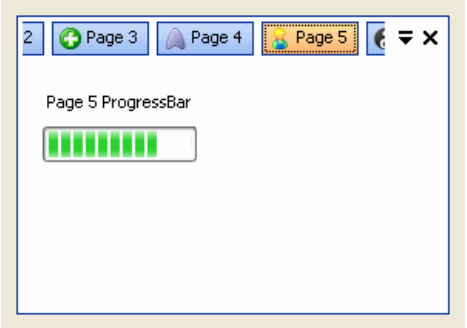
Bar - RibbonTab - Group
A Bar showing Ribbon Tab styled headers is placed on the outside of a Group container. Each Ribbon Tab represents a single Page instance.



Bar - RibbonTab - Only
This is a tab strip mode showing a Ribbon Tab styled header for each Page instance in the navigator.



Bar - CheckButton - Group - Outside
A Bar showing CheckButton controls is placed on the outside of a Group container. Each CheckButton represents a single Page instance.

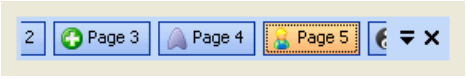


Bar - CheckButton - Group - Inside
A Bar showing CheckButton controls is placed on the inside edge of a Group container. Each CheckButton represents a single Page instance.

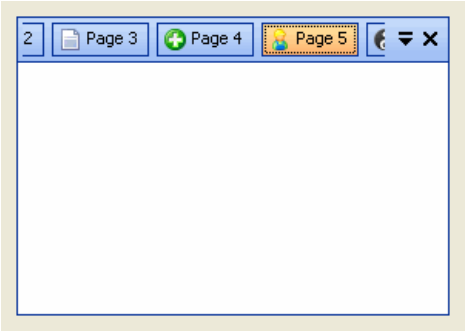
Bar - CheckButton - Group - Only
A Bar showing CheckButton controls is placed inside a Group container. Each CheckButton represents a single Page instance.
This is tab strip style mode and so the contents of the selected page are not displayed inside the Group.

Bar - CheckButton - Only
A Bar is used to show CheckButton

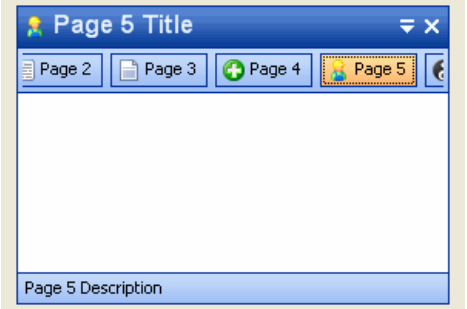
A Bar is used to show CheckButton controls that represent individual Pages. This is a tab strip style mode and so the selected page contents are not displayed within the Navigator.



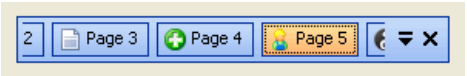
HeaderBar - CheckButton - Group
A Group container with a header that contains a Bar showing CheckButton controls. Each CheckButton represents a single Page instance.



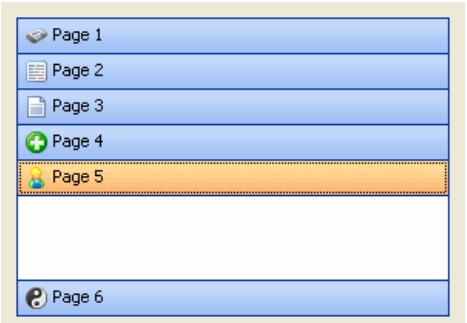
HeaderBar - CheckButton - HeaderGroup
A Group container with two headers for showing information about the currently selected Page.
Plus an additional header that contains a Bar showing CheckButton controls. Each CheckButton represents a single Page instance.



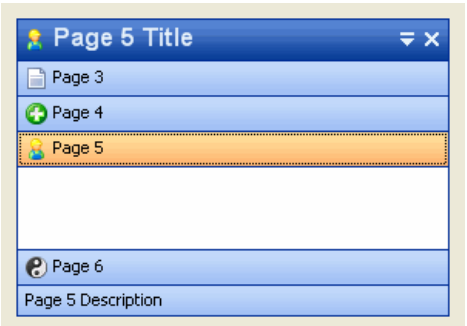
HeaderBar - CheckButton - Only
A header that contains a Bar showing CheckButton controls. Each CheckButton represents a single Page instance.
This is a tab strip style mode and so the selected page contents are not displayed within the Navigator.



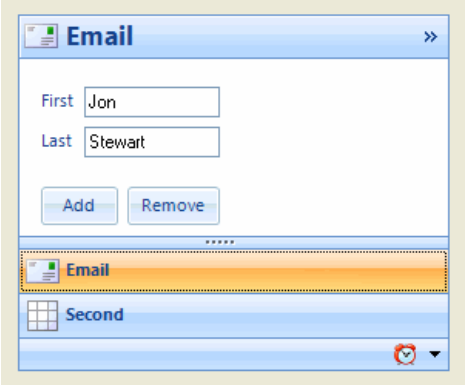
Stack - CheckButton - Group
A vertical or horizontal stack of CheckButton instances. Each CheckButton represents a single Page instance.



Stack - CheckButton - HeaderGroup
A Group container with two headers for showing information about the currently selected Page.
The client area contains either a vertical or horizontal stack of CheckButton instances. Each CheckButton represents a single Page instance.

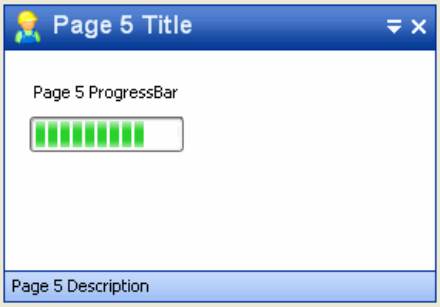


Outlook - Full
Mimics the Microsoft Outlook 2007 style navigation control when in the fully expanded mode.
A stack of CheckButtons is placed at the bottom of the client area. Use the separator to force items onto the overflow bar at the bottom of the control.

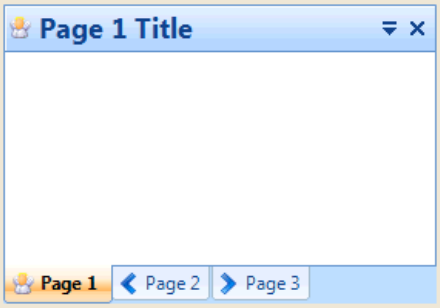


Outlook - Mini
Mimics the Microsoft Outlook 2007 style navigation control where the control is collapsed.
The main client area has button instead of the selected page contents. Click that button to have a pop up window show the actual page contents.

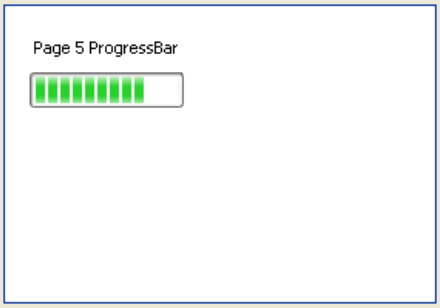
HeaderGroup
A Group container with two headers is used to display information about the currently selected Page.



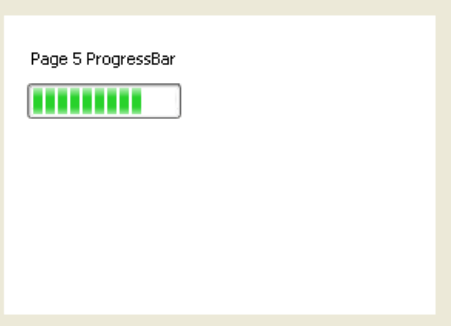
HeaderGroup - Tab
A Group container with two headers is used to display information about the currently selected Page. Combined with a tab area showing the available pages as a set of tabs.



Group
A Group container is used to display the contents of the currently selected Page. The selected Page is sized to fit inside the border of the Group container.
There is no user interface provided for changing the selected page and so any changes must be made using code only.



Panel
A Panel is used to display the contents of the currently selected Page. The selected Page is sized to fit the entire Panel area and so the Panel is not visible if a Page is displayed.
There is no user interface provided for changing the selected page and so any changes must be made using code only.



Navigator Page Dragging

Applicable Properties: AllowPageDrag
DragPageNotify

Applicable Events: BeginPageDrag
AfterPageDrag

AllowPageDrag
A default value of *False* prevents the user from dragging pages away from the Navigator. Note that setting this property on its own is not enough to cause dragging to occur because you also have to set the *DragPageNotify* property so that the drag operation can actually occur.

DragPageNotify
In order to orchestrate a page dragging operation you need to interact with an object that knows how to provide visual feedback and is also aware of the potential drop targets. Any object that provides this ability can be attached to the Navigator by providing the *IDragPageNotify* interface to this property. You do not need to implement this yourself as Krypton comes with a class called *DragManager* that implements the interface and provides palette based feedback drawing for you. For example, to allow two Navigator instances to have pages dragged between them you can use the following simple code:-

```
DragManager dm = new DragManager();

// Add page dragging sources
kryptonNavigator1.DragPageNotify = dm;
kryptonNavigator2.DragPageNotify = dm;

// Add page drop targets
dm.DragTargetProviders.Add(kryptonNavigator1);
dm.DragTargetProviders.Add(kryptonNavigator2);
```

For a more detailed explanation for how page dragging occurs within Krypton you should read the top-level Page Dragging section.

BeginPageDrag, AfterPageDrag
These events are fired before and after the page drag operation. For more details see Other Events.

Navigator PopupPages Properties

Applicable Modes: Bar - Tab - Only
Bar - RibbonTab - Only
Bar - CheckButton - Group - Only
Bar - CheckButton - Only
HeaderBar - CheckButton - Only
Outlook - Mini

PopupPages Properties The set of properties associated with the display of pop up pages can be seen in Figure 1 as they appear in the properties window.

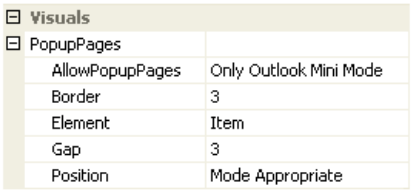


Figure 1 - PopupPages Properties

Pop up page functionality is only available in the modes as listed at the top of this document. These are modes that do not show the contents of the selected page in the layout of the actual navigator control. Commonly these types of mode are called 'TabStrips'. Because the content of the selected page is not displayed in the main part of the control these modes allow the page to be displayed in a separate pop up window when the appropriate button or header is pressed.

AllowPopupPages
Use this enumeration property to decide when pop up pages will be used. The default value of *Only Outlook Mini Mode* only allows the use of pop up pages in the *Outlook - Mini* mode. Other possible values include *Only Compatible Modes* that only allows the use of pop up pages in 'TabStrip' style modes and the *Never* value that prevents pop up pages from appearing at any time.

Border
This is the pixel border width to use around the displayed pop up page. Figure 2 shows a pop up page with the default value of 3, where you can see the thick blue border around the page contents. Figure 3 shows a value of 0 which turns off the use of the blue border area entirely. This border area is displayed using the *Navigator.Panel.PanelBackStyle* defined style.

Figure 2 - Border width of 3

Figure 3 - Border width of 0

Element
When the pop up page is displayed it uses the algorithm specified in the *Position* property to auto calculate the location, and in some cases size, of the pop up window. All the calculations are relative to a display element. You can use this property to define which element is used in the calculations. There are two enumeration values possible for this property, *Item* and *Navigator*. You would use *Item* when you want the pop up window to appear relative to the selected page display item. Alternatively use *Navigator* when you want the pop up to be relative to the whole navigator control instance regardless of the exact location of the page item.

Gap
Use this property to define the pixel gap between the display *Element* and the nearest edge of the pop up window. If you want the pop up window to be shown directly against the *Element* then use a value of 0. The default value is 3 and provides a small spacing gap between the *Element* and the pop up window.

Position
This enumeration property is used to define how the pop up window is sized and positioned relative to the *Element,* taking into

account the *Gap* value. The default value is called *Mode Appropriate* and uses an appropriate *Position* value that depends on the current mode and relevant mode settings. For example, in the *BarTabOnly* mode where the *BarOrientation* is *Top* it will show the pop up below the tab header and aligned to the near edge of the item. In the same mode but with a *BarOrientation* of *Right* it will show the pop up to the near side and aligned to the top of the header.

DismissPopups Method

If you have a button inside a *KryptonPage* and then that page is shown as a pop up then you may want a way to dismiss the pop up when that button is pressed. This is the purpose of the *DismissPopups* method. When the button is pressed you should call this method to ensure that if the *KryptonPage* is displayed inside a pop up then that pop up is dismissed.

Navigator Tooltips Properties

Applicable Modes: Bar - Tab - Group
Bar - Tab - Only Bar - RibbonTab - Group
Bar - RibbonTab - Only Bar - CheckButton - Group - Outside
Bar - CheckButton - Group - Inside Bar - CheckButton - Group - Only
Bar - CheckButton - Only HeaderBar - CheckButton - Group
HeaderBar - CheckButton - HeaderGroup HeaderBar - CheckButton - Only
Stack - CheckButton - Group Stack - CheckButton - HeaderGroup
Outlook - Full Outlook - Mini
HeaderGroup
HeaderGroup - Tab

Tooltips Properties The set of properties associated with the showing of tool tips can be seen in Figure 1 as they appear in the properties window.

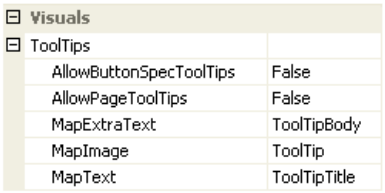


Figure 1 - Tooltips Properties

AllowButtonSpecTooltips
A simple boolean property that determines if tooltips are displayed when the mouse hovers over a button specification. This allows the the pre-defined button specifications such as the *Close/Context/Next/Previous* buttons as well as user defined specification via the *Button.ButtonSpecs* collection property.

AllowPageTooltips
Determines if tooltips are displayed when the mouse hovers over a *KryptonPage* header item. A header item can be a check button, tab header, tab ribbon header, stack item or overflow item depending on the how the mode displays pages. Note that the default value for this property is *False* and so by default no tooltips will be displayed.

MapExtraText + MapText + MapImage
The mapping properties are used to describe how to map the *KryptonPage* values to the tool tip display values.
MapImage is used to recover an image from the page. To prevent any image from being shown assign the *None* value. To show the *ImageSmall* form the page assign *Small* to the *MapImage*. More complex mappings are possible, for example a value of *LargeMediumSmall* indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property. The default is *ToolTip* indicating the *ToolTipImage* property is used.
Use *MapText* to map from the *Text*, *TextTitle, TextDescription, ToolTipTitle* and *ToolTipBody* properties of the page to the main bar item text. The default value is *ToolTipTitle. MapExtraText* defaults to the other tool tip text property *ToolTipBody*.

Navigator Other Properties

Applicable Properties: AllowTabReorder
AllowTabFocus AllowTabSelect
AutoHiddenSlideSize

AllowTabReorder

With a default of *True* this property allows the user to change the order of the pages by using the mouse to drag the page headers to new positions. This works for all the modes that show individual tabs or check buttons for each page that is present.

AllowTabFocus

The default value of *True* causes the Navigator to act like a traditional Windows Forms TabControl. This means that the page header becomes part of the tabbing sequence and this can be seen as you tab around your Form that contains the Navigator. You will notice that the focus stops at the page header before moving on to the first control on the page itself. Clicking on a page header will cause the focus to be placed on the page header. Sometimes this default behavior is not appropriate for your application. It you set the property to False then the page headers do not take the focus. This means that clicking on the page header causes the focus to go straight to the first control on the page itself. Also using Tab will move straight onto the first control on the page and will not stop at the page header. When the Navigator is used inside the Workspace the property is set to be False as this is more appropriate for that environment.

AllowTabSelect

This property determines if any page is allowed to become selected. Use of this property is only recommend in combination with a tab strip style mode. Tab strip modes do not display any page contents and so having no selected page will not cause strange drawing. The main use of this property is within the *KryptonDocking* component where the auto hidden groups are displayed using a *Navigator* with this setting defined.

AutoHiddenSlideSize

Use this property when the page is being used in the *KryptonDocking* component. It defines the size to use for the page when it slides into view from an auto hidden group.

Navigator Bar Modes

Applicable Modes:
Bar - Tab - Group Bar - Tab - Only
Bar - RibbonTab - Group Bar - RibbonTab - Only
Bar - CheckButton - Group - Outside Bar - CheckButton - Group - Inside
Bar - CheckButton - Group - Only Bar - CheckButton - Only

Bar Mode Properties
The set of properties associated with *Bar* and *HeaderBar* modes can be seen below as they appear in the properties window.

Visuals (Modes)	
⊟ Bar	
BarAnimation	True
BarFirstItemInset	0
BarLastItemInset	0
BarMapExtraText	None (Empty string)
BarMapImage	Small
BarMapText	Text - Title
BarMinimumHeight	21
BarMultiline	Singleline
BarOrientation	Top
CheckButtonStyle	Standalone
ItemAlignment	Near
⊞ ItemMaximumSize	200, 200
⊞ ItemMinimumSize	20, 20
ItemOrientation	Auto
ItemSizing	All Same Height
TabBorderStyle	Rounded Outsize Medium
TabStyle	High Profile

Figure 1 - Button Mode Properties

BarAnimation
When the user selects a page that is not fully visible on the bar the page can be scrolled into view using a short animation effect. This is turned on by default and gives a smooth movement to the bar that makes it easier to see how the bar is organized. If you prefer to turn off this feature then set the *BarAnimation* property to *False* and the transition will be instant instead of animated.

BarFirstItemInset + BarLastItemInset The inset distance from the near edge of the bar to the first item is controlled using the BarFirstItemInset property. It defaults to a value of 0 which places the first edge against the control edge. Figure 2 shows two pictures, the left is with the default value of 0 and the right hand picture with a value of 10. Use this property when you need to add the extra spacing gap to improve the look of your application. BarLastItemInset is likewise used to set the inset distance on the far edge and allows you to specify a minimum distance between the last tab and the buttons/control edge.

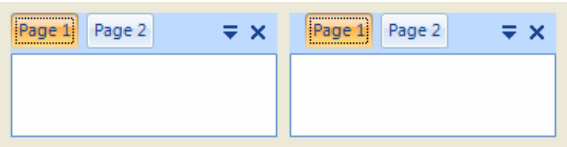


Figure 2 - BarFirstItemInset = 0 & 10

BarMapExtraText + BarMapText + BarMapImage
Each visible *KrytonPage* is represented by a single bar header item, but the *KryptonPage* has three different text properties and three image properties. The mapping properties are used to describe how to map the bar item contents from the values stored in each *KrytonPage* instance.
BarMapImage is used to recover an image from the page. To prevent any image from being shown assign the *None* value. To show the *ImageSmall* form the page assign *Small* to the *MapImage*. More complex mappings are possible, for example a value of *LargeMediumSmall* indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property.
Use *BarMapText* to map from the *Text*, *TextTitle*, *TextDescription* and *TextTooltip* properties of the page to the main bar item text. The default value is *TextTitle* indicating that the *Text* property of the page is used unless it is empty in which case the *TextTitle* property is used. Other variations exist so you can specify a preference for what text is shown. *BarMapExtraText* works in the same way but maps to the secondary bar item text and defaults to *None*.

BarMinimumHeight
The height of the bar area is calculated by discovering the height of the tallest item on the bar. This could be an individual tab header

or one of the action buttons. If there are no items on the bar then the size would become zero height and in order to prevent this from happening you can set the *BarMinimumHeight* property to the smallest height you require.

BarMultiline

Five possible enumeration values determine how items are positioned on lines. The default *Singleline* places all items on a single line that extends beyond the visible area if needed. You can use the *Next/Prev* scroll buttons in order to bring items out of view back into the displayed area. The *Multiline* option will ensure all items are visible by creating as many lines as are required. Use *Exactline* when you need the items to exactly match the line area size, it will expand or shrink the individual items to enforce this. *Shrinkline* will place all items on a single line but also shrink the size of the items if necessary to ensure they are all visible. Finally *Expandline* will only expand the size of items if they do not fill up the entire line display area. Figure 3 shows all five options for the same set of items.

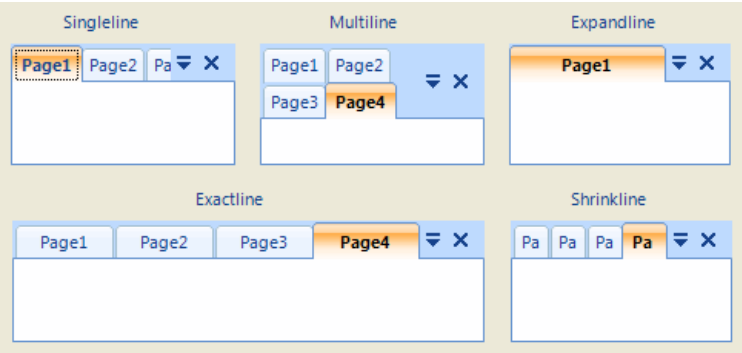


Figure 3 - All possible BarMultiline settings

BarOrientation

By default the bar is positioned at the top of the navigator control as can be seen in Figure 4. You can alter the orientation to any of the three other values *Left*, *Right* and *Bottom*. Figure 5 shows the orientation changed to each of the other options for a range of different *Bar* modes.

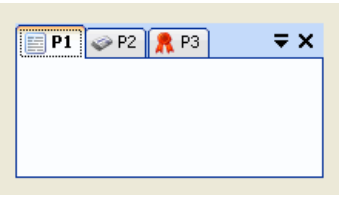


Figure 4 - BarOrientation = Top

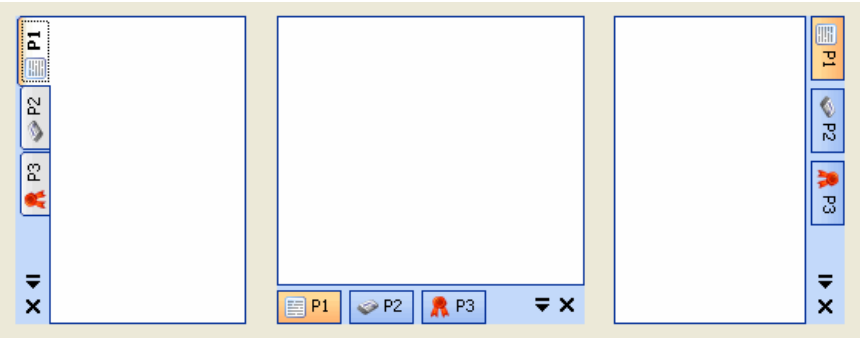


Figure 5 - BarOrientation = Left, Bottom and Right

CheckButtonStyle

When showing the page headers as *CheckButton* items this property is used to specify the button style that should be used for the appearance of each item. The default is the *Standalone* button style but you can change this to any of the other styles such as *LowProfile*, as can be seen in Figure 6.

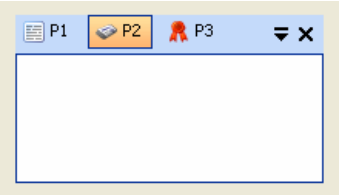


Figure 6 - CheckButtonStyle = LowProfile

ItemAlignment

By default the alignment of items on the bar is to the *Near* side. On a standard western machine this equates to the left hand side of

the control. You can see in Figure 7 and 8 the alternative property values of *Center* and *Far*. The navigator does honor the *RightToLeft* setting and so when defined the *Near* and *Far* values will produce the opposite arrangement.

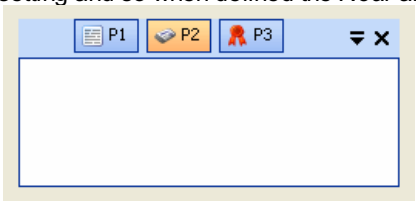


Figure 7 - ItemAlignment = Center

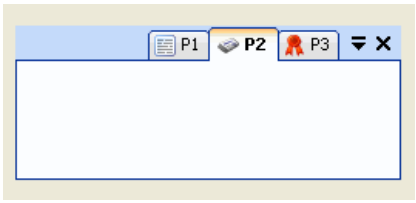


Figure 8 - ItemAlignment = Far

ItemMaximumSize + ItemMinimumSize

Use these two properties to prevent strange looking items under two scenarios. If a header item does not find any text or image to show for a page then it would become sized extremely small, just a few pixels in width and height. To prevent very small items from being created you should use the *ItemMinimumSize* to set a lower limit on header items. The opposite case is where the text and/or image for a page are very large and would create an extremely large header item. Use the *ItemMaximumSize* to set an upper limit on header items.

ItemOrientation

In most cases you will want the orientation of header items to automatically reflect the orientation of the bar itself. So if the bar is at the top then you would want the header items to be orientated as seen in Figure 3. If the bar is placed on the left side then you would want the headers items orientation to show vertical text as seen in Figure 4. This is the purpose of the default *Auto* property value. You can change the property to fix the header orientation. Figure 9 shows the the bar at the top but with the item orientation defined as *FixedLeft*, *FixedBottom* and *FixedRight*.

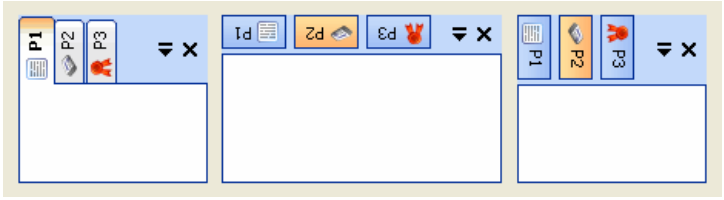


Figure 9 - ItemOrientation = FixedLeft + FixedBottom + FixedRight

ItemSizing

Each bar item is sizing according to the width and height needed to display the contents of that item. The calculated size then has an upper and lower limit applied using the *ItemMaximumSize* and *ItemMinimumSize* properties. The *ItemSizing* property specifies the algorithm to apply across all the bar header items.

- ***Individual Sizing*** Every bar item is left to be whatever size it needs.
- ***All Same Height***
  Once the size of every item has been calculated all of them are set to the same height as the tallest item.
- ***All Same Width*** Once the size of every item has been calculated all of them are set to the same width as the widest item.
- ***All Same Width + Height***
  Combines the *All Same Width* and *All Same Height* options.

TabBorderStyle

Showing the page headers as *Tab* items this property is used to specify the shape of the tab header. The default is the *Rounded Outsize Medium* style but you can change this to any of the other styles such as *OneNote*, as can be seen in Figure 10.

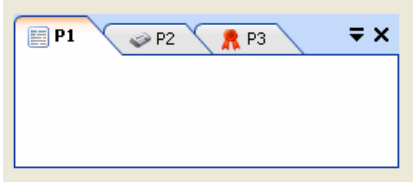


Figure 10 - TabBorderStyle = OneNote

TabStyle
When showing the page headers as *Tab* header items this property is used to specify the tab style that should be used for the appearance of each item. The default is the *High Profile* style but you can change this to any of the other styles such as *LowProfile,* as can be seen in Figure 11.

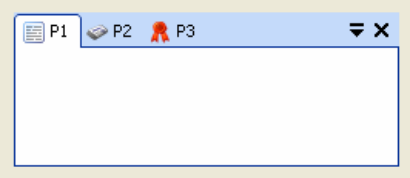


Figure 11 - TabStyle = LowProfile

Navigator Button Modes

Applicable Modes: Bar - Tab - Group
Bar - Tab - Only Bar - RibbonTab - Group
Bar - RibbonTab - Only Bar - CheckButton - Group - Outside
Bar - CheckButton - Group - Inside Bar - CheckButton - Group - Only
Bar - CheckButton - Only HeaderBar - CheckButton - Group
HeaderBar - CheckButton - HeaderGroup HeaderBar - CheckButton - Only
Stack - CheckButton - HeaderGroup Outlook - Full
Outlook - Mini HeaderGroup
HeaderGroup - Tab

Button Mode Properties The properties associated with buttons can be seen in Figure 1 as they appear in the properties window.

Visuals (Modes)	
Button	
ButtonDisplayLogic	Context
ButtonSpecs	**(Collection)**
CloseButton	
CloseButtonAction	RemovePage & Dispose
CloseButtonDisplay	Logic
CloseButtonShortcut	Ctrl+F4
ContextButton	
ContextButtonAction	Select Page
ContextButtonDisplay	Logic
ContextButtonShortcut	Ctrl+Alt+Down
ContextMenuMapImage	Small
ContextMenuMapText	Text - Title
NextButton	
NextButtonAction	Mode Appropriate Action
NextButtonDisplay	Logic
NextButtonShortcut	Ctrl+F6
PreviousButton	
PreviousButtonAction	Mode Appropriate Action
PreviousButtonDisplay	Logic
PreviousButtonShortcut	Ctrl+Shift+F6

Figure 1 - Button Mode Properties

There are four standard buttons exposed for use with the modes listed at the top of the page. These buttons are called *Close*, *Context*, *Next* and *Previous,* each of which has at least three associated properties displayed in Figure 1. So for the *Close* button you can see the three properties listed as *CloseButton*, *CloseButtonAction* and *CloseButtonDisplay*. The *Context* button has a couple of additional properties that will be described after the standard set of three.

Button + ButtonAction + ButtonDisplay + ButtonShortcut
Each button has a property with the extension *Button,* for example *CloseButton* and *NextButton*, that is an aggregate containing many values for defining the appearance of the button. This set of values are not described in detail here as the ButtonSpec section contains a full description of all the properties and how to use them to customize the appearance of the button.

The second extension is *ButtonAction,* for example *CloseButtonAction*, and is used to define the default action that should occur when that button is clicked by the user. When any of the buttons is clicked an event is generated and this property is passed to the event handler as the action to be taken. The event handler can override the action or leave it as the default. Button events are named by adding the word *Action* to the end, for example *NextAction*, *CloseAction* and so forth. A full description of the Action Events is contained in a separate section.
The extension *ButttonDisplay*, for example *CloseButtonDisplay*, is used to specify how to display and enable the button. All of the buttons have a default value of *Logic.* The following list shows the available enumeration values.

- ***Hide***
  This enumeration value will force the button to be hidden from the display.
- ***Show Disabled***
  This value forces the button to always be displayed but disabled.
- ***Show Enabled***
  Forces the button to be displayed always but enabled.
- ***Logic***
  Here the visible and enabled state of the button are determined by runtime logic as specified by the *ButtonDisplayLogic* property. See the section below with the title of *ButtonDisplayLogic Property* for more details.

All except the *Logic* entry allow you to control the visible and enabled state of the button at runtime. This is useful if you want to completely override the operation of a button with your own customization. By altering this property you control the state of the button and then by hooking into the buttons *Action* event you can perform a custom action when the user presses it. More likely is that you will leave this property with the default *Logic* value so that the navigator control handles the runtime state changes for you.

Finally the extension *ButtonShortcut* is used to define the keyboard shortcut that can be used to initiate the button action. For example the *CloseButtonShortcut* property has default value of *Ctrl+F4*, so at runtime the user does not need to use the mouse to click the close button but instead can use the *Ctrl+F4* key combination as a shortcut for invoking the close action. Note that the key combination will only be available if the button itself is available for the navigator mode and the button is visible and enabled.

ButtonDisplayLogic Property
Any button that is defined with a *ButtonDisplay* property value of Logic will use this property to determine the visible and enabled state of the button. The possible values for the property are as follows.

- ***None***
  The *Close* button is always shown but only enabled if a page is selected. The *Context*, *Next* and *Previous* buttons are never displayed.
- ***Next/Previous***
  The *Close* button is always shown but only enabled if a page is selected. The *Context* button is never displayed. The Next and *Previous* buttons are always displayed but only enabled if the action they represent is possible.
- ***Context***
  The *Close* button is always shown but only enabled if a page is selected. The *Context* button is always shown but only enabled if at least one page is visible. The *Next* and *Previous* buttons are never displayed.
- ***Context + Next/Previous***
  Combines the *Next/Previous* and *Context* options. The *Context*, Next and Previous buttons are always displayed but only enabled if they can perform their actions. Figure 2 shows this enumeration value in operation.

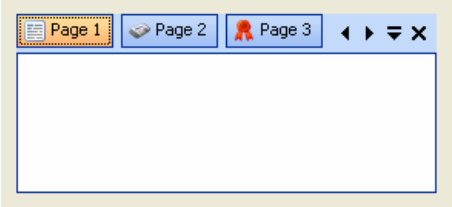


Figure 2 - ContextButton

ContextMenuMapImage + ContextMenuMapText
These two properties are used to describe how to map values from a *KryptonPage* to the image and text of a context menu item. Figure 3 shows how the *Context* button shows a context menu that presents a menu item per page that can be selected. As a *KryptonPage* has three different text values and three different images the navigator needs to know to pull the correct values from the page to each menu item.

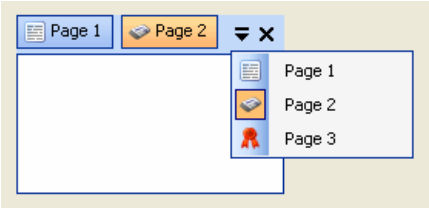


Figure 3 - ContextButton

The default value for the *ContextMenuMapImage* is *Small* and indicates that the *ImageSmall* property of the *KryptonPage* will be used for the menu item image. If you prefer to show the *ImageMedium* or *ImageLarge* from the page then just assign the *Small* and *Large* values respectively instead. To prevent any image from being shown then use the *None* value. You can also specify a value of *LargeMediumSmall* that indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property. Other variations exist so you can specify the list of preferences for the image to use.
Using *ContextMenuMapText* is much the same except it is working against the *Text*, *TextTitle*, *TextDescription* and *TextTooltip* properties of the *KryptonPage*. The default value is *TextTitle* indicating that the *Text* property of the page is used unless it is empty in which case the *TextTitle* property is used. Other variations exist so you can specify a preference for which text properties to use.

ButtonSpecs
This collection property allows you to add your own buttons to the display. See the ButtonSpec section for a full description of how to create and configure a *ButtonSpec* for use. Figure 4 shows an example of a bar mode that has a custom button added.

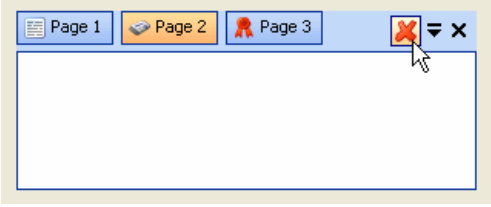


Figure 4 - Custom button using ButtonSpecs

Navigator Group Modes

Applicable Modes: Bar - Tab - Group
Bar - RibbonTab - Group Bar - CheckButton - Group - Outside
Bar - CheckButton - Group - Inside Bar - CheckButton - Group - Only
HeaderBar - CheckButton - Group HeaderBar - CheckButton - HeaderGroup
Stack - CheckButton - HeaderGroup Outlook - Full
Outlook - Mini HeaderGroup
HeaderGroup - Tab
Group

Group Mode Properties

The set of properties associated with *Group* modes can be seen in Figure 1 as they appear in the properties window.

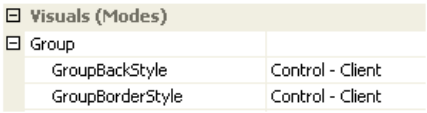


Figure 1 - Group Mode Properties

Several navigator modes display a *Group* as part of the appearance. You can use the *GroupBackStyle* and *GroupBorderStyle* in order to alter the background and border styles respectively for the *Group* appearance. By default both properties are defined as *Control - Client* and so you get the effect shown in Figure 2 below, where the *Bar - CheckButton - Group - Outside* mode is used but with no pages defined. Figure 3 shows both properties changed to *Header - Primary.*

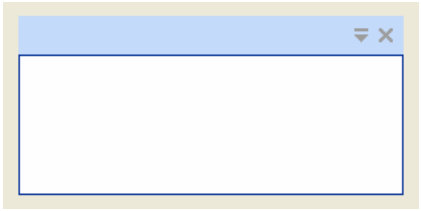


Figure 2 - Mode = Bar - CheckButton - Group - Outside

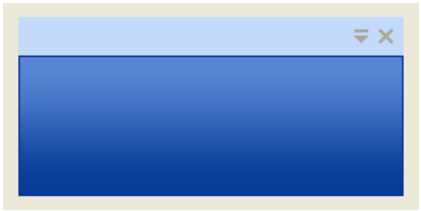


Figure 3 - Styles = Header - Primary

Navigator Header Modes

Applicable Modes: HeaderBar - CheckButton - HeaderGroup
Stack - CheckButton - HeaderGroup Outlook - Full
Outlook - Mini HeaderGroup
HeaderGroup - Tab

Header Mode Properties
The set of properties associated with Header related modes can be seen in Figure 1 as they appear in the properties window.

Visuals (Modes)	
Header	
HeaderPositionBar	Top
HeaderPositionPrimary	Top
HeaderPositionSecondary	Bottom
HeaderStyleBar	Secondary
HeaderStylePrimary	Primary
HeaderStyleSecondary	Secondary
HeaderValuesPrimary	
Description	
Heading	(Empty)
Image	(none)
ImageTransparentColor	
MapDescription	None (Empty string)
MapHeading	Title - Text
MapImage	Small - Medium
HeaderValuesSecondary	
HeaderVisibleBar	True
HeaderVisiblePrimary	True
HeaderVisibleSecondary	True

Figure 1 - Header Mode Properties

Header Positions
The primary, secondary and bar headers can be positioned against any of the form control edges. To alter the location of the primary header alter the *HeaderPositionPrimary* to an alternative value such as *Left*, *Right* or *Bottom*. Use the *HeaderPositionSecondary* property to modify the location of the secondary header and *HeaderPositionBar* for the bar header. Figure 2 shows the *HeaderGroup* mode with default settings and Figure 3 with the primary header on the *Left* and the secondary header on the *Right*.

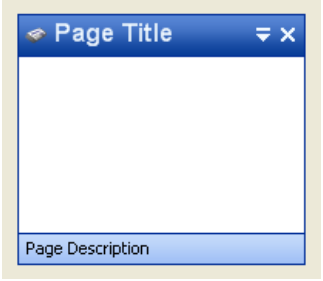


Figure 2 - Default HeaderGroup appearance

Figure 3 - Modified header positions

Header Styles
The header style can be altered using the *HeaderStylePrimary*, *HeaderStyleSecondary*, *HeaderStyleBar* properties. Figure 2 shows the default appearance with the primary header using the primary header style and the secondary header the secondary header style. Figure 4 shows an example where the *HeaderStylePrimary* property has been altered to *Secondary*.

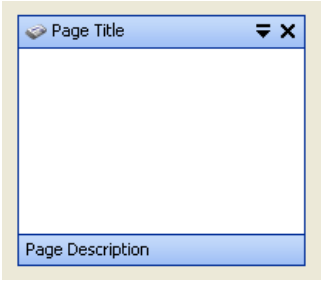


Figure 4 - HeaderStylePrimary = Secondary

Header Values

The *HeaderValuesPrimary* set of properties are used to determine the text and image to show on the primary header. Likewise the *HeaderValuesSecondary* property set determines the display for the secondary header. Note that there is no property for the bar header as the bar header contains per page controls whose content are determined in the *Bar* set of visual properties.

Each set contains a *Header, Description* and *Image* property that are used to define the display when there is no page selected. These are useful because when the navigator does not have any visible pages you will want to define what is shown on the header. Showing a message to indicate the empty state prevents the user becoming confused because there is no text displayed at all.

The *MapHeader*, *MapDescription* and *MapImage* properties are used when a page is selected and determine how to extract values from the selected *KryptonPage* for display. *MapImage* is used to recover an image from the *KryptonPage*. To prevent any image from being shown assign the *None* value. To show the *ImageSmall* form the page assign *Small* to the *MapImage*. More complex mappings are possible, for example a value of *LargeMediumSmall* indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property.

Use *MapHeader* to map from the *Text*, *TextTitle*, *TextDescription* and *TextTooltip* properties of the *KryptonPage* to the main header text. The default value is *TitleText* indicating that the *TextTitle* property of the page is used unless it is empty in which case the *Text* property is used. Other variations exist so you can specify a preference for what text is shown. *MapDescription* works in the same way but maps to the header description text and defaults to *None*.

Header Visibility

All the headers can be removed from display. Use the *HeaderVisiblePrimary, HeaderVisibleSecondary* and *HeaderVisibleBar* boolean properties to change the visibility. Note that if you remove the primary header that contains the *Close*, *Context* and other buttons then those buttons will not longer be accessible to the user.

Navigator Outlook Modes

Applicable Modes: Outlook - Full
Outlook - Mini

Outlook Mode Properties

The set of properties associated with the *Outlook* modes can be seen in Figure 1 as they appear in the properties window.

Visuals (Modes)	
Outlook	
BorderEdgeStyle	Control - Client
CheckButtonStyle	Navigator Stack
Full	
HeaderSecondaryVisible	False
ItemOrientation	Auto
Mini	
Orientation	Vertical
OverflowButtonStyle	Navigator Overflow
ShowDropDownButton	True
TextAddRemoveButtons	&Add or Remove Buttons
TextFewerButtons	Show Fe&wer Buttons
TextMoreButtons	Show &More Buttons

Figure 1 - Outlook Mode Properties

BorderEdgeStyle

A border edge is drawn between each of the stack items in order to provide a consistent looking border around all elements of the control. If you are altering the associated *CheckButtonStyle* then you will likely also want to alter this property so that the border edge drawing is consistent with the updated check button style.

CheckButtonStyle

By default this property has a value of *Navigator Stack* that is defined specifically for use with the stack modes of the navigator control. You can however change to any of the other button styles such as *Standalone*, *LowProfile*, *Custom1* etc. It is recommended that you use one of the custom styles if you need to alter how you display the stack items and use the *Palette Designer* application to setup the properties of the custom style.

HeaderSecondaryVisible

Other modes determine the visibility of the secondary header using the *KryptonNavigator.Headers.HeaderVisibleSecondary*. But when in the *Outlook* mode this property is used instead. This is because you will almost certainly want the header visibility to differ between the *Outlook* and other modes. If however you do want to use the same visibility for all modes then just set this property to be *Inherit*.

ItemOrientation

In most cases you will want the orientation of stack content to automatically reflect a sensible default related to the orientation of the stack itself. So if the stack is vertical you would want the stack content drawn horizontally and when the stack is horizontal the stack content would be drawn vertically. This is the purpose of the default *Auto* property value. You can change this property to fix the content orientation to a constant setting.

Orientation

You can alter the default *Vertical* orientation to *Horizontal* and you can see the change in figure 2. Notice that although the stacking items and the overflow bar are repositioned the primary header is not moved.

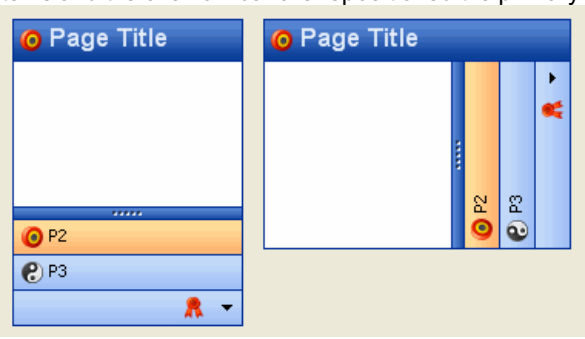


Figure 2, Orientation = Vertical and Horizontal

OverflowButtonStyle

By default this property has a value of *Navigator Overflow*that is defined specifically for use in the overflow bar at the bottom of the outlook modes. The pages that are on the overflow bar as well as the drop down button are drawn using this style.

ShowDropDownButton
If you want to prevent the drop down button from appearing on the overflow bar then you just set this property to be *False*.

TextAddRemoveButtons + TextFewerButtons + TextMoreButtons
When you click the drop down button on the overflow bar you will see the menu as shown in figure 3. You can use these the three text properties in alter the text that is displayed for each of the three menu items. This does not change the functionality of the menu items but it allows you to localize the displayed string because the three properties are marked with the *Localizable* attribute.

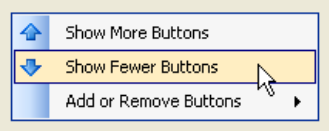


Figure 3, Drop down button menu
To customize the contents of the drop down menu you should hook into the *KryptonNavigator.OutlookDropDown* event. The event provides a reference to the *ContextMenuStrip* that is about to be displayed, this allows you to add/remove/modify the menu items as you pleas.

Outlook Full - Mode Properties
The set of properties that are specific to the *Outlook Full* mode are contained below the *Full* property that is listed in Figure 1 above. Figure 2 shows the child properties where the *Full* property is expanded inside the properties window.

Full	
OverflowMapExtraText	None (Empty string)
OverflowMapImage	Small
OverflowMapText	None (Empty string)
StackMapExtraText	None (Empty string)
StackMapImage	Medium - Small
StackMapText	Text - Title

Figure 2 - Outlook Full - Mode Properties

OverflowMapExtraText + OverflowMapImage + OverflowText
The overflow mapping properties are used to map between KryptonPage values and entries on the overflow bar.
OverflowMapImage is used to recover an image from the page. To prevent any image from being shown assign the *None* value. To show the *ImageSmall* form the page assign *Small* to the *MapImage*. More complex mappings are possible, for example a value of *LargeMediumSmall* indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property.
Use *OverflowMapText* to map from the *Text*, *TextTitle, TextDescription* and *TextTooltip* properties of the page to the main overflow item text. The default value is *TextTitle* indicating that the *Text* property of the page is used unless it is empty in which case the *TextTitle* property is used. Other variations exist so you can specify a preference for what text is shown. Stack*MapExtraText* works in the same way but maps to the secondary overflow text and defaults to *None*.

StackMapExtraText + StackMapImage + StackMapText The stack mapping properties are used to map between KryptonPage values and the stack item contents.

StackMapImage is used to recover an image from the page. To prevent any image from being shown assign the *None* value. To show the *ImageSmall* form the page assign *Small* to the *MapImage*. More complex mappings are possible, for example a value of *LargeMediumSmall* indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property.
Use *StackMapText* to map from the *Text*, *TextTitle*, *TextDescription* and *TextTooltip* properties of the page to the main stack item text. The default value is *TextTitle* indicating that the *Text* property of the page is used unless it is empty in which case the *TextTitle* property is used. Other variations exist so you can specify a preference for what text is shown. *StackMapExtraText* works in the same way but maps to the secondary stack item text and defaults to *None*.
Outlook Mini - Mode Properties
The set of properties that are specific to the *Outlook Mini* mode are contained below the *Mini* property that is listed in Figure 1 above. Figure 2 shows the child properties where the *Mini* property is expanded inside the properties window.

Mini	
MiniButtonStyle	Navigator Mini
MiniMapExtraText	None (Empty string)
MiniMapImage	None (Null image)
MiniMapText	Text - Title
StackMapExtraText	None (Empty string)
StackMapImage	Medium - Small
StackMapText	None (Empty string)

Figure 3 - Outlook Mini - Mode Properties

MiniButtonStyle
In *Outlook Mini* mode you will see that the client area of the layout does not show the contents of the selected KryptonPage. Instead the area is filled with a button that has the name of the page. When clicking that button it will then show a pop up window with the

contents of the associated KryptonPage. This property is used to define the button style that is applied to the drawing of that client area button.

MiniMapExtraText + MiniMapImage + MiniMapText
The mini mapping properties are used to map between KryptonPage values and the mini button used in the client area of the control. *MiniMapImage* is used to recover an image from the page. To prevent any image from being shown assign the *None* value. To show the *ImageSmall* form the page assign *Small* to the *MapImage*. More complex mappings are possible, for example a value of *LargeMediumSmall* indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property.
Use *MiniMapText* to map from the *Text*, *TextTitle* and *TextDescription* properties of the page to the main stack item text. The default value is *TextTitle* indicating that the *Text* property of the page is used unless it is empty in which case the *TextTitle* property is used. Other variations exist so you can specify a preference for what text is shown. *MiniMapExtraText* works in the same way but maps to the secondary stack item text and defaults to *None*.

StackMapExtraText + StackMapImage + StackMapText
The stack mapping properties are used to map between KryptonPage values and the stack item contents. See the above description of the properties.

Navigator Panel Modes

Applicable Modes: Bar - Tab - Group
Bar - Tab - Only Bar - RibbonTab - Group
Bar - RibbonTab - Only Bar - CheckButton - Group - Outside
Bar - CheckButton - Only
HeaderBar - CheckButton - Only
Panel

Panel Mode Properties The set of properties associated with Panel modes can be seen in Figure 1 as they appear in the properties window.

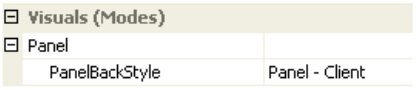


Figure 1 - Panel Mode Properties

The only property you can alter is called *PanelBackStyle* and alters the background style of the panel area within the navigator appearance. A default value of *Panel - Client* can be seen in the above picture but you can alter this to any of the available background style values. When the control is in the *Panel* mode the entire client area of the navigator control takes on the *PanelBackStyle* appearance. However, this mode fills the entire client area with the currently selected page and so only if there are no visible pages in the navigator will you see this background.

When using one of the *Bar* modes listed above the background of the bar area uses the *PanelBackStyle*. Changing the *PanelBackStyle* will change the background of the bar area to the requested style. Figure 2 shows the *Bar - CheckButton - Group - Outside* mode and the light blue area behind the check buttons and bar buttons is the area affected by the *PanelBackStyle* property.

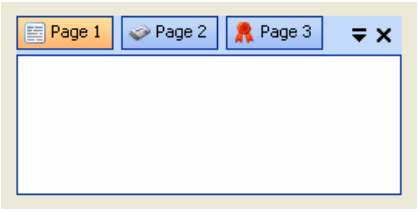


Figure 2 - Mode = Bar - CheckButton - Group - Outside

Navigator Stack Modes

Applicable Modes: Stack - CheckButton - Group
Stack - CheckButton - HeaderGroup

Stack Mode Properties

The set of properties associated with *Stack* modes can be seen in Figure 1 as they appear in the properties window.

Visuals (Modes)	
Stack	
BorderEdgeStyle	Control - Client
CheckButtonStyle	Navigator Stack
ItemOrientation	Auto
StackAlignment	Center
StackAnimation	True
StackMapExtraText	None (Empty string)
StackMapImage	Small
StackMapText	Text - Title
StackOrientation	Vertical

Figure 1 - Stack Mode Properties

BorderEdgeStyle

A border edge is drawn between each of the stack items in order to provide a consistent looking border around all elements of the control. If you are altering the associated *CheckButtonStyle* then you will likely also want to alter this property so that the border edge drawing is consistent with the updated check button style.

CheckButtonStyle

By default this property has a value of *Navigator Stack* which is defined specifically for use with the stack modes of the navigator control. You can however change to any of the button styles such as *Standalone*, *LowProfile*, *Custom1* etc. It is recommended that you use one of the custom styles if you need to alter how you display the stack items and the *Palette Designer* application to setup the properties of the custom style.

ItemOrientation

In most cases you will want the orientation of stack content to automatically reflect a sensible default related to the orientation of the stack itself. So if the stack is vertical you would want the stack content drawn horizontally and when stack is horizontal the stack content would be drawn vertically. This is the purpose of the default *Auto* property value. You can change this property to a constant setting so the content orientation is fixed in both stack orientations. Figure 2 shows the the item orientation defined as *FixedLeft*, *FixedBottom* and *FixedRight*.

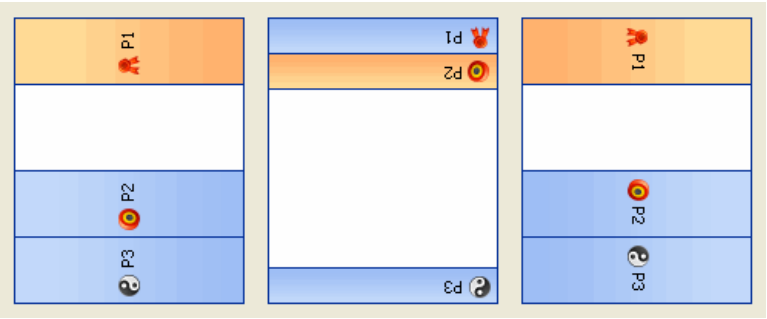


Figure 2 - ItemOrientation = FixedLeft + FixedBottom + FixedRight

StackAlignment

Use this property to define the positioning of the stack items relative to the client area. So a value of *Near* will place all the stack items at the top of the client area and the contents of the selected page at the bottom. Using the *Far* setting will place all the stack items at the bottom of the client area and the selected page contents at the top. The default setting of *Center* will the stack items before the selected page at the top and the stack items after the selected page at the bottom. You can see each of the three settings in figure 3.

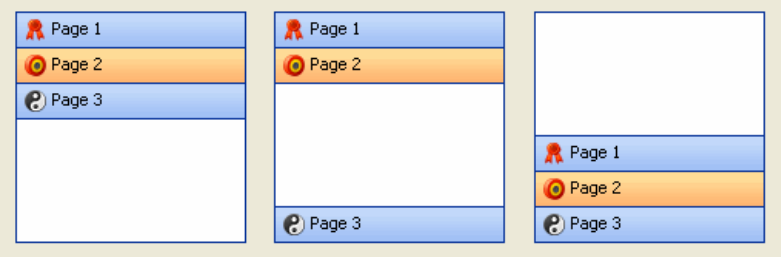


Figure 3, StackAlignment = Near, Center and Far

StackAnimation

When the user selects a page that is not fully visible the page can be scrolled into view using a short animation effect. This is turned on by default and gives a smooth movement to the scrolling that makes it easier to see how the stack items are organized. If you prefer to turn off this feature then set the Stack*Animation* property to *False* and the transition will be instant instead of animated.

StackMapExtraText + StackMapImage + StackMapText

Each visible *KrytonPage* is represented by a single stack item, but the *KryptonPage* has three different text properties and three image properties. The mapping properties are used to describe how to map the stack item contents from the values stored in each *KrytonPage* instance.

StackMapImage is used to recover an image from the page. To prevent any image from being shown assign the *None* value. To show the *ImageSmall* form the page assign *Small* to the *MapImage*. More complex mappings are possible, for example a value of *LargeMediumSmall* indicates that the *ImageLarge* property of the page should be used unless it is null, in that case use the *ImageMedium* property instead but if that is also null then use the *ImageSmall* page property.

Use *StackMapText* to map from the *Text*, *TextTitle*, *TextDescription* and *TextTooltip* properties of the page to the main stack item text. The default value is *TextTitle* indicating that the *Text* property of the page is used unless it is empty in which case the *TextTitle* property is used. Other variations exist so you can specify a preference for what text is shown. Stack*MapExtraText* works in the same way but maps to the secondary stack item text and defaults to *None*.

StackOrientation

You have two orientation options, the default of *Vertical* and the alternative of *Horizontal*. You can see both of these options presented Figure 4. Note that when used with on a *Form* that has the *RightToLeftLayout* and *RightToLeft* settings defined as *True* it will reverse the ordering of the *Horizontal* orientation stack items.

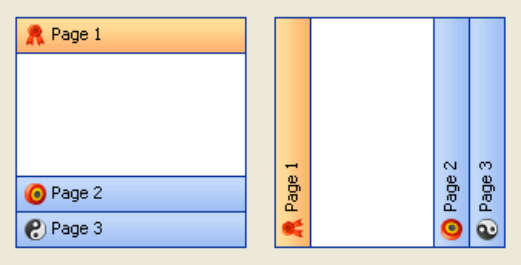


Figure 4, StackOrientation = Vertical and Horizontal

Navigator Selection Events

Applicable Events: Deselecting
Selecting Deselected
Selected SelectedPageChanged

Standard Event Sequence

When a change in selected page is requested the following sequence of events occur. Note that this sequences occurs when either the user initiates a change or when you programmatically change selection by altering the *SelectedIndex* or *SelectedPage* properties of the navigator control. There is never any difference in the events generated or the ordering of events just because the source of the change is programmatic compared to user interaction.

- ***Deselecting***
  Generated for the currently selected page in order to ask if the page is allowed to be deselected. This event can be cancelled so if the event handler sets the *e.Cancel* value to *True* then the selection change is aborted immediately. Use this event to perform page level validation and decide if you will allow the page to be deselected.
- ***Selecting***
  Generated for the page that is to become selected and asks if the page is allowed to be selected. This event can be cancelled so if the event handler sets the *e.Cancel* value to *True* the the selection change is aborted immediately. Use this event to decide if the page is capable of becoming selected.
- ***Deselected*** Generated for the currently selected page and indicates that it is now becoming deselected. This event cannot be cancelled and so you should use the event to perform any cleanup actions that are required when the page is no longer to be displayed.
- ***Selected*** Generated for the page that is to now becoming selected. This event cannot be cancelled and so you should perform actions to initialize the new page ready for display as the new selection.
- ***SelectedPageChanged***
  Generated when the value of the *SelectedPage* property is changed to reflect the new selection. This event cannot be cancelled and is always the last event in the sequence. If you do not need to perform any page specific actions such as initialization, cleanup or validation then this is the most appropriate event to monitor for selection changes.

You can use the *Basic Events* example that is found in the *Navigator* section of the *Krypton Explorer* to see the events that are generated during selection changes. This handy example allows you to add and remove pages to the navigator and lists the selection events that occur as you click around the pages.

First Page Sequence

When the first visible page is added the sequence of events differs from that listed above. There is obviously no existing page that needs to be deselected and so no need to generate the *Deselecting* and *Deselected* events. So the generated event sequence becomes the following.

- ***Selecting***
  Generated for the page that is to become selected. Any attempt to cancel the event will be ignored because one of the visible pages must be selected at all times. Although the event cannot be cancelled it is still fired so you can be sure that the logic inside the handler is always called before the page becomes selected.
- ***Selected*** Generated for the page just added to the control. This event cannot be cancelled and so you should perform actions to initialize the new page ready for display as the new selection.
- ***SelectedPageChanged***
  Generated when the value of the *SelectedPage* property is changed to reflect the new selection. This event cannot be cancelled and is always the last event in the sequence. If you do not need to perform any page specific actions such as initialization, cleanup or validation then this is the most appropriate event to monitor for selection changes.

Last Page Sequence

When the last visible page is removed from the navigator only two events are fired. Once the last page is removed there is no choice left but to set the selected page to *(null)*. Therefore the only two events fired are as follows.

- ***Deselected*** Generated for the currently selected page and indicates that it is now becoming deselected. This event cannot be cancelled and so you should use the event to perform any cleanup actions that are required when the page is no longer to be displayed.
- ***SelectedPageChanged***
  Generated when the value of the *SelectedPage* property is changed to reflect the new selection. This event cannot be cancelled and is always the last event in the sequence. If you do not need to perform any page specific actions such as initialization, cleanup or validation then this is the most appropriate event to monitor for selection changes.

Remove Page Sequence

This sequence is applied when removing the selected page and there are one or more other visible pages still present. With other pages still present one of them will have to become the new selection but you can use the *Selecting* event to determine which one it will be. You cannot specify the actual page you would like to become selected in the any of the generated events, but you can keep cancelling the *Selecting* event for all the pages except the one you want to become selected.

There is no *Deselecting* event because the page is being removed from the *KryptonNavigator.Pages* collection and you cannot

prevent the page from being removed by cancelling the event. Therefore the first event fired is *Deselected*. This is then followed by firing the *Selecting* event for all the remaining visible pages. If one of the *Selecting* events is not cancelled then it becomes the new selection. If all the remaining pages cancel the event then the control will automatically select the first enabled page that it tested. When there are no enabled pages remaining then it chooses the first disabled page that was tested.

- ***Deselected*** Generated for the currently selected page and indicates that it is being removed and so becoming deselected. This event cannot be cancelled and so you should use the event to perform any cleanup actions that are required when the page is no longer to be displayed.
- ***Selecting***
  This event is fired for all remaining visible pages. If the event is not cancelled then it becomes the new selection. If all the remaining pages cancel the event then the control will automatically select the first enabled page that it tested. When there are no enabled pages remaining then it chooses the first disabled page that was tested. Note that default value of the *e.Cancel* property is *True* if the page is disable and *False* if the page is enabled. This ensures that if you do not override the event then the default action for the control will be to selected the next available enabled page.
- ***Selected***
  Generated for the page that is to now becoming selected. This event cannot be cancelled and so you should perform actions to initialize the new page ready for display as the new selection.
- ***SelectedPageChanged***
  Generated when the value of the *SelectedPage* property is changed to reflect the new selection. This event cannot be cancelled and is always the last event in the sequence. If you do not need to perform any page specific actions such as initialization, cleanup or validation then this is the most appropriate event to monitor for selection changes.

KryptonPage.Visible Property
Just because a page has been added to the navigator does not mean it has to be displayed. You can use the *KryptonPage.Visible* property and the *KryptonPage.Hide()* and *KryptonPage.Show()* methods to alter the page visibility at any time. As far as the active selection and the selection events are concerned a page that is not visible does not exist.
If you have a single page in the *KryptonNavigator.Pages* collection that is currently hidden and then you set that page to be visible then the *First Page Sequence* will occur, because as far as the navigator is concerned this is the first page that is now capable of being displayed and selected. If you then hide that page the *Last Page Sequence* will be fired.
So although a page is might be constantly present in the *KryptonNavigator.Pages* collection you should think of it as only being present for selection purposes when set to be visible.

KryptonPage.Enabled Property Unlike the visible property the enabled property does not always prevent a page from becoming selected. Under normal circumstances a disabled page will not become selected because the user interface will not allow the user to select that page. But there are circumstances that will cause a disabled page to be selected.

If the only page displayed is in the disabled state then it will be selected. Remember that one of the visible pages must be in the selected state. This can happen when you add just a single page to the navigator that is also disabled. Alternatively if you remove all pages from the control except a disabled one then it will become selected. Also if you change the currently selected page from enabled to disabled then it will retain the selected state.

Navigator Action Events

Applicable Events: NextAction
PreviousAction CloseAction
ContextAction

NextAction Event
This event is generated when the user presses the *NextAction* button that is presented for some of the navigator modes. The event is generated in order to request information about the appropriate action that should be taken by the mode. By default the value of the *e.Action* event property is assigned from the *KryptonNavigator.Button.NextButtonAction* control property. You can alter this event property inside the event handler to modify the action you would like to take place. Available actions are listed below.

- ***None***
  The navigator will perform no action. This is useful if you want to implement some custom processing for the *NextAction* button. Just assign this value to the *e.Action* event property and then perform whatever application specific logic you require.
- ***SelectPage***
  The navigator control will select the next available visible page.
- ***MoveBar***
  The navigator will scroll the display bar to show the next set of page headers. Note this action will have no effect unless the current navigator mode is showing a bar with page headers displayed. Even then it will only perform a scroll if the bar is not already at the end of the set of page headers and so there are more pages headers that can be brought into view.
- ***ModeAppropriateAction***
  The navigator will use the action most appropriate for the currently selected navigator mode. The bar modes will use the MoveBar action and the HeaderGroup mode the SelectPage action.

PreviousAction Event
Generated when the user presses the *PreviousAction* button this event is used to request information about the appropriate action that should be taken by the mode. Note that the *PreviousAction* button is only presented for some of the navigator modes. By default the value of the *e.Action* event property is assigned from the *KryptonNavigator.Button.PreviousButtonAction* control property. You can alter this event property inside the event handler to modify the action you would like to take place. Available actions are listed below.

- ***None***
  The navigator will perform no action. This is useful if you want to implement some custom processing for the *PreviousAction* button. Just assign this value to the *e.Action* event property and then perform whatever application specific logic you require.
- ***SelectPage*** The navigator control will select the previous available visible page.
- ***MoveBar*** The navigator will scroll the display bar to show the previous set of page headers. Note this action will have no effect unless the current navigator mode is showing a bar with page headers displayed. Even then it will only perform a scroll if the bar is not already at the start of the set of page headers and so there are more pages headers that can be brought into view.
- ***ModeAppropriateAction***
  The navigator will use the action most appropriate for the currently selected navigator mode. The bar modes will use the *MoveBar* action and the *HeaderGroup* mode the *SelectPage* action.

Close Action Event
Pressing the *CloseAction* button generates this event which is used to request information about the appropriate action that should be taken. Note that the *CloseAction* button is only presented for some of the navigator modes. By default the value of the *e.Action* event property is assigned from the *KryptonNavigator.Button.CloseButtonAction* control property. You can alter this event property inside the event handler to modify the action you would like to take place. Available actions are listed below.

- ***None***
  The navigator will perform no action. This is useful if you want to implement some custom processing for the *CloseAction* button. Just assign this value to the *e.Action* event property and then perform whatever application specific logic you require.
- ***HidePage***
  The navigator control will set the *KryptonPage.Visible* property of the selected page to be *False*. Useful if you need to hide pages so they can be made visible again later on.
- ***RemovePage***
  The navigator will remove the selected page from the *KryptonNavigator.Pages* collection. Note that it does not call *KryptonPage.Dispose()* method and so the page can be added back to the same or a different navigator control in the future.
- ***RemovePageAndDispose***
  The navigator will perform the *RemovePage* action as above and also call the *KryptonPage.Dispose()* method on the page. You should not try and add the page back to a navigator in the future as the page has been disposed.

ContextAction Event
This event is used to allow customization of the displayed *KryptonContextMenu* as well as requesting the action to take for the selected context menu item. Note that the *ContextAction* button is only presented for some of the navigator modes.
By default the event will provide a *KryptonContextMenu* instance in the *e.KryptonContextMenu* event property. There will be a single

menu item in the context menu strip for each page that is allowed to be selected. If you need to customize the context menu by adding, removing or modifying entries then this is the appropriate place to do so. If you add extra context menu items then you need to add event handlers for processing their selection as the navigator control will ignore them should the user select one.
The *e.Action* event property indicates the action to take if the user selects one of the auto generated context menu items. By default the value of the *e.Action* event property is assigned from the *KryptonNavigator.Button.ContextButtonAction* control property. You can alter this event property inside the event handler to modify the action you would like to take place. Available actions are listed below.

- ***None*** The navigator will perform no action.
- ***SelectPage***
  The navigator control will select the page that the user selected from the context menu strip.

Navigator Other Events

Applicable Events: TabCountChanged
TabVisibleCountChanged TabDoubleClicked
TabMouseHoverStart TabMouseHoverEnd
PrimaryHeaderLeftClicked PrimaryHeaderRightClicked
PrimaryHeaderDoubleClicked DisplayPopupPage
OutlookDropDown ShowContextMenu
BeginPageDrag AfterPageDrag
PageDrop CtrlTabStart
CtrlTabWrap
TabClicked

TabCountChanged
Fired when the number of pages in the control has changed.
TabVisibleCountChanged
Occurs when the number of visible pages has changed. This can occur when the number of actual pages remains constant but the *Visible* state of the pages is updated.
TabDoubleClicked
When the user double clicks a tab with the left mouse button this event is fired.
TabMouseHoverStart
If the user moves the mouse over a tab and leaves the mouse static for a short period of time then this event if fired. This is useful if you want to display your own tool tip style feedback.
TabMouseHoverEnd
Generated when the mouse moves after a TabMouseHoverStart has been created. These two events are always paired.
PrimaryHeaderLeftClicked
Fired when the user left mouse clicks on a primary header element.
PrimaryHeaderRightClicked
Fired when the user right mouse clicks on a primary header element.
PrimaryHeaderDoubleClicked
When the user double clicks with the left mouse button on a primary header element.
DisplayPopupPage
This event is fired whenever a pop up page is about to be shown. It has a *Cancel* property that allows you to prevent the pop up page from being shown by setting the *Cancel* to be *True*. It also has a *ScreenRectangle* parameter that you can inspect and modify to change the screen location and size of the pop up. It will be populated with a calculated value that is used unless you decide to override the value. Also provided is an *Item* property that gives you a reference to the *KryptonPage* that is going to be shown.
OutlookDropDown
When using either the *Outlook - Full* or the *Outlook - Mini* mode there is a small drop down arrow available in the overflow area of the control. When you click this drop down this event is generated. It has a property called *KryptonContextMenu* that contains the set of menu items for display to the user. You can modify this list by adding your own custom items that you would like presented to the user.
ShowContextMenu
If you right-click a page header then this event is generated as an attempt to show a page specific context menu. By default the events *e.Cancel* property is defined as *True* so that no context menu is shown. If you would like to show a context menu for the page then you must set this property to *False* in the event handler.

The event arguments have two properties called *ContextMenuStrip* and *KryptonContextMenu* that are used to specify the context menu to be shown. Use of *KryptonContextMenu* takes precedence. When the event is generated the value of the context menu properties is defaulted to the settings from the actual *KryptonPage*. So the event argument *KryptonContextMenu* property is defined as the value of *KryptonPage.KryptonContextMenu* and the event argument *ContextMenuStrip* is defined as the *KryptonPage.ContextMenuStrip* value.

BeginPageDrag
Just before page dragging occurs this event is fired to allow event handlers a chance to either cancel the operation or modify the set of pages that are being dragged. Note that *AllowPageDrag* must be set to *True* before any page dragging can occur and so this event will not be fired unless that property is set. Once this event has finished and has not been cancelled the *PageDragNotify* interface, if present, will be used during the drag operation for providing feedback.
AfterPageDrag
After the dragging operation has completed this event is fired so the event handler can cleanup and reverse any actions they took during the *BeginPageDrap*. It provides a property indicating if the drop was completed with success, so the drop occurred, or if the operation was cancelled.
PageDrop
This event is fired when a page is being dropped into the navigator instance. You can use this event to cancel the drop or alter the provided page reference in order to alter the page actually dropped.
CtrlTabStart
When the user presses the *Ctrl+Tab* key combination this event is fired so that you can cancel the event and prevent the normal action from occuring. Normally this key combination will cause the selected page to shift to the next visible and enabled page. *Ctrl+Shift+Tab* also fires the event and selected the previous page instead of the next page. One of the event arguments indicates if the shift key is pressed.

CtrlTabWrap
The *Ctrl+Tab* key combination causes the next appropriate page to become selected. If the searching reaches the end of the page list it wraps around to the first page again to continue the search. Hence using the *Ctrl+Tab* combination rotates around all the pages continually. To prevent this wrapping around of the page list you hook into this event and set the *Cancel* event argument. One of the event arguments indicates if the shift key is pressed.
TabClicked
This event is generated when the user left mouse clicks on a page tab header. Tab headers including the ribbon tabs, standard tabs and the check buttons that are used for many of the available modes. This event is generated even if the page is already the selected page.

Workspace

Learn more about the capabilities of the *KryptonWorkspace* using the following sections.

Sections

- Overview
- Layout
- Compacting
- Sizing
- Persistence
- Page Dragging
- Events

Workspace Overview

Workspace Properties

The *KryptonWorkspace* specific properties are shown in Figure 1.

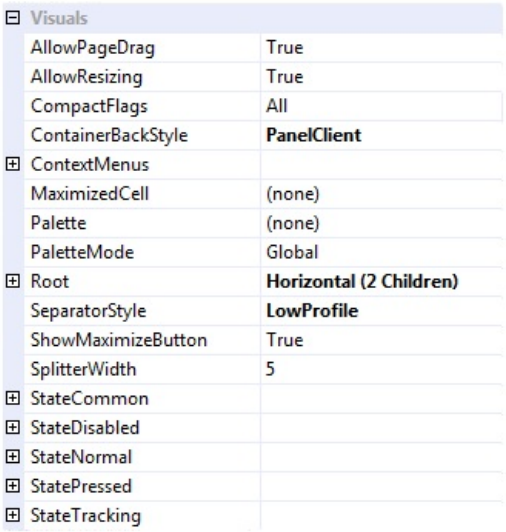


Figure 1 - Workspace properties

AllowPageDrag
A boolean property that determines if the user is allowed to drag pages in order to change the workspace layout.

AllowResizing
Determines if the user is allowed to resize cells by moving the separators that are positioned between cells. Also note that there is an *AllowResizing* property on each of the *KryptonWorkspaceCell* instances that can prevent resizing on a per-cell basis. In order to move a separator this property must be *True* as well as the cell on on either side of the separator.

CompactFlags
Just before the workspace is sized and positioned a compacting phase is run to optimize the structure of the workspace hierarchy definition. This set of flags determine the compacting actions that are allowed to take place. For more detail read the Compacting section that describes reach of the flags.

ContainerBackStyle
Use this style to determine the appearance of the workspace background. The background can be seen when you have no contents visible or between the cells when the separator style chooses not to draw any separator element.

ContextMenus
This is a collection of properties that specify the display text and keyboard shortcuts used for the context menu that appears when you right click a page header. Try right click on a page header in the workspace and a context menu is shown with a set of options for modifying the layout. For example you can use *Move Next* and *Move Previous* to transfer the page to the next/previous cell.

MaximizedCell
Assign a KryptonWorkspaceCell that exists inside the workspace hierarchy to order to have that cell maximized and showing as the only cell in the workspace client area. This is useful if you want to allow the user to concentrate on a single cell for a period of time. Set the property to *null* to remove the maximized setting.

Palette, PaletteMode
These properties allow you to define the palette for use when drawing the control. By default it will use the global palette as defined by the *KryptonManager* instance.

Root
This is a *KryptonWorkspaceSequence* instance that represents the starting point for defining the workspace definition. You can add *KryptonWorkspaceCell* and *KryptonWorkspaceSequence* instances into a sequence in order to create a tree like hierarchy. For more details you should read the Layout section followed by the related Sizing section.

SeparatorStyle
Style used for drawing the separators that exist between the individual cell entries.

ShowMaximizeButtin
Use this property to determine if an extra button should be added to each workspace cell that is used to toggle between maximized and restored states.

SplitterWidth
Pixel width of the separators that exist between the individual cell entries.

Four States
The separator can be in one of four possible states, *Disabled*, *Normal*, *Tracking* and *Pressed*. When resizing is allowed for the separator it will be in the *Normal* state until the user moves the mouse over the separator area at which point it enters the *Tracking* state. If the user presses the left mouse button whilst over the separator then it enters the *Pressed* state. If the workspace control has been disabled then each separator is also placed in the *Disabled* state.

Common State
To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define the border width in *StateNormal* and *StateCommon* then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*..

Workspace Layout

Cells and Sequences

You can create any complexity of workspace layout by combining the *KryptonWorkspaceCell* and *KryptonWorkspaceSequences* elements. Each *KryptonWorkspaceCell* represents a leaf node in the layout and is actually a class derived from the *KryptonNavigator*. Leaf nodes are those that do not have any children. As each cell is actually a specialized *Navigator* instance you can customize the appearance with any of the *Navigator* capabilities. The *KryptonWorkspaceSquence* is used to contain a set of child elements and by combing sequences within sequences you can create a tree like structure that defines the layout hierarchy.

KryptonWorkspace.Root

The starting point for defining the layout is the *Root* property of the *KryptonWorkspace* that is actually just a *KryptonWorkspaceSequence* instance. Each sequence, including the *Root*, has a property called *Orientation* that determines the direction that child elements are positioned. To demonstrate Figure 1 has a *Root* sequence with three cells inside. The left image has a *Root.Orientation* of *Horizontal* and the right image a value of *Vertical*.

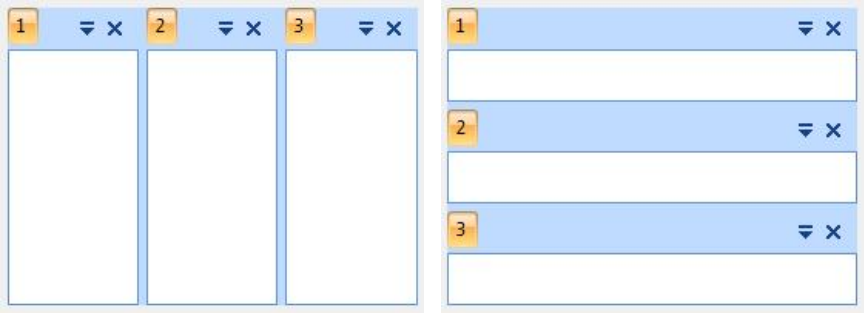


Figure 1 - Horizontal and Vertical orientations

Children within a sequence are always sized to fill the opposite direction. So in the left image we have a horizontal sequence where each cell is automatically defined to be the full height of the area. The height definition of the cell is ignored as the height is auto generated. The opposite is true of the right image. With the sequence being vertical the width is automatically defined to fill the entire width of the sequence and the cell defined width value ignored. The following hierarchy represents the left hand image:-

```
Sequence  (Horizontal)         Cell (Page 1)
     Cell (Page 2)
     Cell (Page 3)
```

Embedded Sequences

To create a more complex layout we need to embed sequence instances inside the root sequence. Figure 2 shows a simple design where we have a cell on the left and then on the right we have two cells that are arranged in a vertically column.

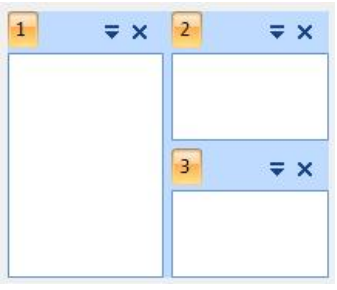


Figure 2 - Sequence with the Root sequence

The hierarchy of elements looks like the following with the first sequence being the *Root* sequence property of the *KryptonWorkspace*.

```
Sequence       (Horizontal)         Cell      (Page 1)
     Sequence  (Vertical)              Cell (Page 2)
          Cell (Page 3)
```

We can keep going and place another sequence in the place of the third cell above. In that case we would achieve Figure 3.

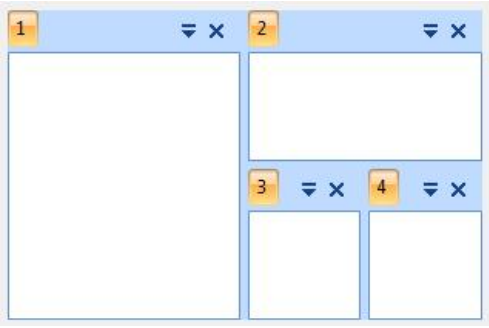


Figure 3 - Sequence within Sequence within Root sequence

This hierarchy of elements now looks like the following.

```
Sequence            (Horizontal)          Cell            (Page 1)
     Sequence       (Vertical)               Cell       (Page 2)
          Sequence  (Horizontal)                      Cell (Page 3)
               Cell (Page 4)
```

Perform Layout

You can make as many changes as you like the workspace hierarchy but no change will occur until the control performs a layout cycle. This happens when you explicitly call the *PerformLayout* method or when the next windows message is processed. So you are safe to make multiple changes to the workspace definition without the changes actually occuring until windows messages are processed again. As cells are added and removed from the child collection of controls only during the layout processing it means you will not receive the *WorkspacePageAdding* and *WorkspacePageRemoved* events during your changes to the workspace definition. Instead they will occur some time afterwards once layout processing is performed by the control.

Compacting

As the user drags and drops pages around the workspace extra sequences are created in order to create the appropriate appearance. After many such changes the workspace definition would become very disorganized and inefficient. In order to prevent this a compacting phase is performed just before the layout. For this reason you will notice that if you add a sequence that has no children the sequence is automatically removed from the definition because it is redundant. To see a full list of the compacting actions you are recommend to read the compacting section.

Workspace Compacting

Compacting Flags

One of the unique features of the KryptonWorkspace is the hierarchical structure of the layout definition. It means you can quickly and easily create complex layouts for your document area. But dragging pages around could cause the structure to become inefficient and leave redundant entries. To prevent this happening there are some flags that indicate how to perform a compacting phase just before the layout of the contents occur. Each of the compact flags is used to resolve a potential inefficiency. The flags are exposed via the *KryptonWorkspace.CompactFlags* property. Each of the flags is explained below with an example of how they change the layout definition.

RemoveEmptyCells

Using drag and drop you can transfer a page from one cell to another. If you remove the last page in the cell you are left with a cell that has no pages. Some applications may want this to happen but many others would prefer the cell to be automatically removed as an empty cell is redundant. When this flag is set the compacting phase will search for any cells that have no pages defined and remove them from the definition.

RemoveEmptySequences

Removing empty cells can result in a workspace sequence existing that has no child elements. Once all the cells contained in the sequence have been removed we are left with a redundant sequence. The user will not see the sequence as it is likely to be zero sized but it still exists in the hierarchy. During compacting this flag will search for sequences that have no children and automatically remove them.

PromoteLeafs

A more subtle scenario involves a sequence that contains just a single child. If a sequence contains just a single child item then that sequence itself is redundant as it only encloses one item. This flag will promote the single child into the place of the sequence and delete the no longer needed sequence. This situation occurs if you used drag and drop to move pages around. You can end up with a sequence that contains a single sequence child that contains yet another single sequence child and so forth. Although you would not see any visual impact it is clearly inefficient to have long chains of single sequences.

AtLeastOneVisibleCell

Use this flag to ensure you always have at least one visible cell in the workspace. If the user deletes the last showing page in the last showing cell then the *RemoveEmptyCells* flag above would remove the last cell and leave the workspace area blank. Some applications require that there is always at least one cell showing even if it has no pages in it. This flag will ensure that is the case.

Example

Imagine we have the following workspace definition in place when compacting occurs.

```
Sequence          (Horizontal)          Sequence    (Vertical)
            Cell  (1 Page)
      Sequence    (Vertical)                Cell  (No Pages)
            Cell  (No Pages)
```

RemoveEmptyCells removes empty cells giving.

```
Sequence          (Horizontal)          Sequence    (Vertical)
            Cell  (1 Page)
      Sequence    (Vertical)
```

RemoveEmptySequences removes empty sequences giving.

```
Sequence       (Horizontal)          Sequence (Vertical)
         Cell  (1 Page)
```

PromoteLeafs replaces sequences with a single item giving.

```
Sequence       (Horizontal)          Cell     (1 Page)
```

Workspace Sizing

StarSize Property

Both the *KryptonWorkspaceCell* and the *KryptonWorkspaceSequence* have a property called *StarSize* that is used to define the sizing for that part of the workspace layout. You can define the cell/sequence to have either a fixed size or a proportional sizing value. The *StarSize* is a string type that is expected to contain two values separated by a comma. The 'Star' in *StarSize* refers to the notation used when providing a proportional sizing value. This is best explained using a series of pictures. Figure shows two cells that are both defined to have a width of '1*'.

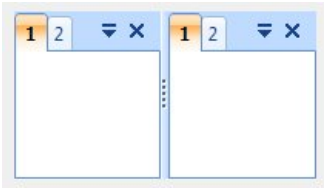


Figure 1 - Two cells with '1' as the width*

The available width is first allocated to whatever fixed size cells are defined. We have none in this example and so the entire width is then allocated to the cells with stars defined. The total number of stars is 2 and so each cell gets half the available width as they each have half the number of total stars. As we increase the size of the Workspace control we get the following change in appearance as seen in Figure 2.

Figure 2 - Two cells with '1' as the width*

Both still get half the total width. If we update the first cell to have a value of '3*' for the width we can see Figure 3 showing the resultant change.

Figure 3 - Two cells with '3' and '1*' for the widths*

The total number of stars is 4 and so the first cell is given 3/4 of the space and the second cell just 1/4. Using star sizing makes it very easy for you to indicate the relative proportions of each cell. These examples only show the width changing but the height works in the same way when the cells are placed in a vertical sequence. Note that in a horizontal sequence the width is defined by using the width of the *StarSize* and in a vertical sequence the height is defined from the height of the *StarSize*. The opposite direction from the sequence is always defined as whatever is needed to fill sequence. So in our above examples we are using a horizontal sequence and so the height of each cell is always the entire height of the sequence itself.

Fixed and Proportional

Most real world scenarios will likely include fixed as well as proportional cells. To show this we define three cell with widths defined as '100', '1*' and '1*' respectively. Figure 4 shows how this will appear at runtime.

Figure 4 - Three cells with '100', '1' and '1*' for the widths*

The first cell is fixed in width and so allocated the full 100 pixels with the remaining space split evenly between the other two cells. If we expand the Workspace to be be wider we get the change as seen in Figure 5.

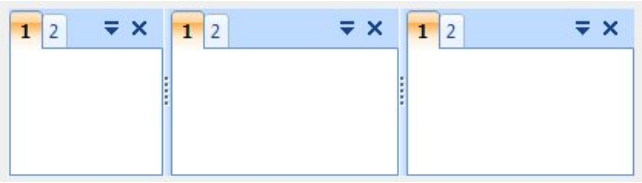


Figure 5 - Three cells with '100', '1' and '1*' for the widths*

As expected the first cell has stayed the same fixed width and the others have split up the larger space between themselves. If we change the width of the third cell to be '4*' we get the Figure 6 appearance.

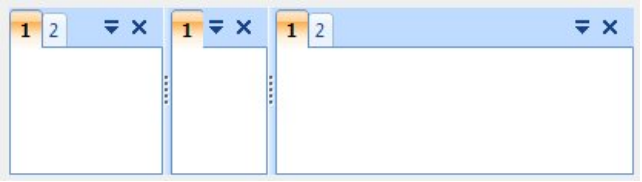


Figure 6 - Three cells with '100', '1' and '4*' for the widths*

The last cell takes up 4/5 of the remainder space and the second cell just 1/5. Shrinking the window changes the appearance as seen in Figure 7.

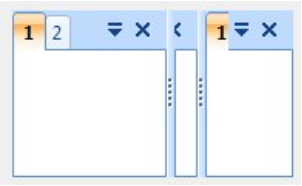


Figure 7 - Three cells with '100', '1' and '4*' for the widths*

The combination of fixed and proportional sizing gives the flexibility you need to construct just about any scenario you are likely to need. When you use the splitter to change th sizing it will automatically update the star and fixed sizes correctly to reflect the change. Calling the size fixed only refers to a fixed layout and does not mean the users cannot change the size using the splitter. You prevent a cell from being resized by the user you should set the *KryptonWorkspaceCell.AllowResizing* property to *False*.

Workspace Persistence

Applicable Methods:

SaveLayoutToArray, LoadLayoutFromArray SaveLayoutToFile, LoadLayoutFromFile
SaveLayoutToXml, LoadLayoutFromXml
SaveLayoutToStream, LoadLayoutFromStream

Persistence Formats

You can save and load the workspace layout in a variety of different formats in order to suit the needs of your application. Use the *SaveLayoutToFile* and *LoadLayoutFromFile* methods in order to persist to files in the XML format. This is useful if you need to retain a layout when an application exits in order to restore it again when next run. Alternatively you can use the *SaveLayoutToArray* and *LoadLayoutFromArray* pair of methods that persist as an array of bytes. This makes it easy to place the data into a database, transfer it over a network connection or just store it within your application. For greater control you can use the *SaveLayoutToXml* and *LoadLayoutFromXml* pair that expect *XmlTextWriter* and *XmlTextReader* instances. Finally the last pair *SaveLayoutToStream* and *LoadLayoutFromStream* provide maximum flexibility because they accept a generic stream for the storage. This allows you to easily integrate the configuration information into your own persistence mechanism.

Information Stored

The hierarchy of the workspace is saved which consists of the tree of workspace sequences and workspace cells. On loading it will remove the existing hierarchy and create a new one that matches the loaded configuration. For each cell it also stores the list of pages that it contains. On loading it will either create a new page to match the one that was saved or actually reuse the existing page if it still exists in the workspace. It determines if the same page exists by comparing the *UniqueName* of the page that was saved with the *UniqueName* of all the existing pages.

When an existing page matches the incoming name it uses the existing page rather than create an entirely new page. This is ideal when you want to rearrange the existing set of pages as you can save and later reload the layout and it will rearrange those pages to the saved organization. Ensure you use consistent *KryptonPage.UniqueName* values for the pages so that the saved information continues to match the current pages.

In a dynamic scenario you will have different sets of pages over time and so you will not always have the saved page already present in the workspace. In this case the loading will not find a match and so create a new page instance. It will restore the basic information about the page including the text, tool tip and image values. But it does not persist the set of child controls that exist on the page or the visual information such as modified *StateCommon* values. To recreate the set of child controls and any other page specific information you need to hook into the *PageSaving* and *PageLoading* events.

Global Custom Data

You may need to store additional application specific data along with the layout configuration for use when reloading. You can do this quite easily by hooking into the *GlobalSaving* and *GlobalLoading* events. The saving event will provide an *XmlWriter* reference that should be used to save your extra information. Create additional XML elements with whatever information you need to persist. Loading provides an *XmlReader* that can be used to traverse and load back that same information.

We recommend saving your own version number into the custom data so that in the future you can recognize a change in the way you have stored the data. This makes it easier to change the storage and then still be able to recognize older formats and be able to process them.

Page Custom Data

Handling per-page custom data is similar to the above global custom data method. Use the *PageSaving* and *PageLoading* events to hook into the process and use the event parameters to get a reference to the actual page that is being saved/loaded. You are given a *XmlWriter* for saving and *XmlReader* for loading the data. An extra feature of the loading event is the ability to modify the page reference provided as an event parameter and have that new page reference used instead of the instance passed into the event. This allows you to override the loading process and force the use of your own designated page. If you override the page reference with *null* then you will prevent any page being added at all. So you can dynamically decide if a loading page is desired and use *null* to reject it from being added into the workspace.

Workspace Page Dragging

Applicable Properties: AllowPageDrag
DragPageNotify

Applicable Events: BeginPageDrag
AfterPageDrag

AllowPageDrag
A default value of *True* allows the user to drag pages around the workspace in order to modify the layout of the contents. You can drag individual page headers such as the tabs and check buttons that represents pages in a typical cell appearance. If the cell has been modified to display in a header mode such as *HeaderGroup* then you can drag all the pages it contains in one go by dragging the primary header of the cell. Setting this property to *False* will prevent any user dragging from occuring.

DragPageNotify
When this property is left as the default *null* you can only drag and drop pages within the individual Workspace instance. In order to combine this control with other page dragging enabled controls you need to explicitly assign a *IDragPageNotify* interface to this property. Krypton provides an implementation of this interface called *DragManager* and it allows you to link together multiple drag sources and targets. For example, to allow two Workspace instances to have pages dragged between them you can use the following simple code:-

```
DragManager dm = new DragManager();

// Add page dragging sources
kryptonWorkspace1.DragPageNotify = dm;
kryptonWorkspace2.DragPageNotify = dm;

// Add page drop targets
dm.DragTargetProviders.Add(kryptonWorkspace1);
dm.DragTargetProviders.Add(kryptonWorkspace2);
```

For a more detailed explanation for how page dragging occurs within Krypton you should read the top-level Page Dragging section.

BeginPageDrag, AfterPageDrag
These events are fired before and after the page drag operation. For more details see Events.

Workspace Events

Applicable Events: WorkspaceCellAdding
WorkspaceCellRemoved ActiveCellChanged
ActivePageChanged
MaximizedCellChanged BeginPageDrag
AfterPageDrag PageDrop
GlobalSaving GlobalLoading
PageSaving PageLoading
RecreateLoadingPage PagesUnmatched
CellCountChanged CellVisibleCountChanged

WorkspaceCellAdding
Each time a new *KryptonWorkspaceCell* is added to the workspace controls collection this event is fired so that you can customize the settings of that cell. As the cell is a class derived from the *KryptonNavigator* you might want to consult the Navigator documentation to see the full range of available modes and appearance options. As well as changing the appearance you are recommend to use this event for attaching to any cell events. You should use the *WorkspaceCellRemoved* event to unhook from those events you attach to here. This is required so that there are no references to the cell once it is removed and so it is eligable for garbage collection.

WorkspaceCellRemoved
Fired when the *KryptonWorkspaceCell* instance is removed from the controls collection. Use this event to unhook from any events you attached to in the *WorkspaceCellAdding* event. If the cell has the *DisposeOnRemove* property defined as *True* then the cell will have been disposed before this event is generated.

ActiveCellChanged
This event is fired each time the value of the *ActiveCell* property changes. This can happens in several ways, most commonly when the user selects a page on a different cell. It also occurs if you drag and drop a page to create a new cell or when you update the layout using a method such as *ApplySingleCell*. Use this event if you want to alter the appearance of the active cell so the user can more easily see which cell is active within the workspace layout. For example you might change the tab drawing style so the active cell is more prominent.

ActivePageChanged
Generated when the value of the *ActivePage* property changes. Each time the user selects a new page using the mouse or a keyboard combination the property is updated.

MaximizedCellChanged
Fired when the value of the *MaximizedCell* property changes. Note that this property can change from user interactions and not just because of a programmatic change to the property.

BeginPageDrag
Just before page dragging occurs this event is fired to allow event handlers a chance to either cancel the operation or modify the set of pages that are being dragged. Note that *AllowPageDrag* must be set to *True* before any page dragging can occur and so this event will not be fired unless that property is set. Once this event has finished and has not been cancelled the *PageDragNotify* interface, if present, will be used during the drag operation for providing feedback.

AfterPageDrag
After the dragging operation has completed this event is fired so the event handler can cleanup and reverse any actions they took during the *BeginPageDrap*. It provides a property indicating if the drop was completed with success, so the drop occurred, or if the operation was cancelled.

PageDrop
This event is fired when a page is being dropped into the navigator instance. You can use this event to cancel the drop or alter the provided page reference in order to alter the page actually dropped.

GlobalSaving
Called during the save layout process and allows custom data to be added into the persisted data. You are provided with an *XmlWriter* reference that should be used for saving your information. Custom data can be structured by adding new elements and attributes as needed so that the XML is structured in a logical way.

GlobalLoading
Called during the load layout process and allows previously saved data to be handled. An *XmlReader* reference is provided and

should be used to navigate and process the incoming information.

PageSaving
Each time a page is saved this event is called and provided with a reference to the page along with an *XmlWriter*. Use the text writer to save any additional information you require associated with the page.

PageLoading
Each time a page is loaded this event is fired and provided a reference to the page along with an *XmlReader*. Load additional information using the text reader and then perform page setup actions such as creating controls for the page. You can override the page reference in order to change the page that will be added to the workspace. So you can create an entirely new page and modify the event page reference so that the new page you just created is used in place of the one provided. If you modify the event page reference to be *null* then the load process will not add any page to the workspace. This is useful if your application needs to reject the loading of individual pages.

RecreateLoadingPage
During the load process the incoming pages are matched against pages that already exist inside the workspace. If the incoming page has the same unique name as an existing page then the existing page is used and loaded with the incoming details and then positioned appropriately. If the incoming page does not already exist in the workspace then this event is fired so that the page can be recreated. At the end of the event processing the page details will be loaded into the provided page and then added into the workspace.

PagesUnmatched
Fired at the very end of the loading process this event provides a list of pages that were present in the workspace before the load occurred but were not matched by and loading page details. Without further action these unmatched pages will be removed from the workspace. Use this event if you want to prevent some or all of these pages being removed. You must use code to add them back into the workspace hierarchy and into a cell in order for them to be retained not automatically removed.

CellCountChanged
At the end of the workspace layout phase this event is fired if the number of cells within the workspace has changed since the last layout phase. Firing the event at the end of a layout allows the developer to make multiple changes to the hierarchy of the control without each individual operation causing the event to be fired. Instead the event will be fired only if the aggregate of the operations results in a changed value for the number of cells once the layout process has finished.

CellVisibleCountChanged
If the number of visible cells has changed at the end of the control layout phase then this event is fired. Note that the number of actual cells might be constant but if the number of the cells that are visible changes then the event is still fired. Conversely if the number of cells changes but the number of those visible is the same then the event will not be fired.

Page Dragging

Learn more about dragging and dropping *KryptonPage* instances using the following sections.

Topics

- Overview
- DragManager
- Drag Enabling Controls

Page Dragging

Krypton provides its own drag and drop mechanism for moving *KryptonPage* instances between compatible controls. Note this is entirely separate from the standard drag and drop system that is exposed in Windows Forms using *DoDragDrop* and *IDropTarget*. Why provide a separate mechanism when windows already has a standard approach? Krypton provides more sophisticated visual feedback to the user that would be very difficult to achieve using the standard approach. It also allows us to provide something that is designed to be easy to use with Krypton controls.

Overview
You need three elements to perform page dragging. First you need a set of drop targets and for that we have the *IDragTargetProvider*. Any control that wants to act as a drop target needs to expose *IDragTargetProvider* so the drop targets for that control can be discovered. Second you need to allow drag operations to be initiated and for that we have the *IDragPageNotify* interface. Any control that wants to start a drag operation needs to take an instance of that *IDragPageNotify* interface and call its methods as required. Finally we need to an object that can orchestrate the operation. Krypton provides a class called *DragManager* that performs this orchestration for you.

IDragTargetProvider
Any control that needs to act as a drop target needs to expose this simple interface. The interface only has a single method called *GenerateDragTargets* that is called each time the set of drop targets is needed. Note that this means it is called each time a page drag operation is started so you could return a different set of targets for each drag operation. Passed into the method is a reference to the *PageDragEndData* instance which contains a list of all the pages being dragged. So you can examine the set of pages being dragged and decide which targets are relevant to those pages. Returned from the *GenerateDragTargets* call is a list of targets so you can provide none, one or many.

The Navigator implements this interface by returning just a single drop target. When the user drops on that target all the dragged pages are added to the end of the Navigator page list. Workspace also implements this interface but returns many different targets. The Workspace has many potential drop points such as the workspace edges or edges for each individual workspace cell and so the returned list of targets could be quite extensive. When the drop occurs the relevant target then performs the appropriate action.

For more detailed information see the Drag Enabling Controls section.

IDragPageNotify
Any control that needs to initiate dragging needs to do so my making calls into this interface. The interface exposes several methods such as *PageDragStart*, *PageDragMove*, *PageDragEnd* and *PageDragQuit* that are called by the source control as various actions occur. When the source control notices the left mouse button being pressed it would call *PageDragStart*. It would then call *PageDragMove* as the mouse is moved up until the release of the left button at which point *PageDragEnd* is called. If at any time the operation needs to be aborted then *PageDragQuit* is used.

Note that the control does not implement this interface but only makes calls into it. In order to do this the control needs to be provided with an instance of the *IDragPageNotify* interface. Both the Navigator and Workspace controls expose the *DragPageNotify* property for this very purpose. You can provide the same instance of the *IDragPageNotify* interface to more than one control as only one control at a time can be performing a drag operation.

For more detailed information see the Drag Enabling Controls section.

DragManager
Use the *DragManager* to orchestrate page dragging with the targets and provide visual feedback during the operation. Once you have created an instance of the class you can attach target providers by using the *DragTargetProviders* collection. For example you can add Navigator and Workspace instances that implement the IDragTargetProvider interface in the following way:-

```
DragManager dm = new DragManager();

dm.DragTargetProviders.Add(kryptonNavigator1);
dm.DragTargetProviders.Add(kryptonWorkspace1);
```

The *DragManager* also implements the *IDragPageNotify* interface and so can be attached directly to the Navigator and Workspace controls like so:-

```
kryptonNavigator1.DragPageNotify = dm;
kryptonWorkspace1.DragPageNotify = dm;
```

For more detailed information see the DragManager section.

DragManager

If you want to enable dragging of *KryptonPages* between different controls you need to create an instance of the *DragManager* class. This class links together a set of target providers so that they are all available during a drag operation. Using the class is very easy. First of all create an instance using the default constructor. Second you add to the *DragTargetProviders* collection each of the dragging targets. Third and last you assign the *DragManager* to the *DragPageNotify* property of each control that can start a drag.

Properties

DragTargetProviders
This collection is used to store a list of *IDragTargetProvider* instances. A call to *DragStart* begins the dragging process and this list is used to generate the drop targets for use during the entire drag operation.

Palette
If you provide a *KryptonPalette* instance to this property it will be used as the source for appearance values when drawing the visual feedback.

PaletteMode
A default of *GlobalPalette* causes the global palette defined by the *KryptonManager* to be used for appearance values. If you provide a *Palette* property value then this will be updated to *Custom* to reflect the use of a directly provided palette instance. To alter values for just this *DragManager* you should update the values contained within the *StateCommon* property.

StateCommon
Overrides for altering the appearance of the drag feedback. The most significant of these is called *Feedback* and is an enumeration of possible feedback drawing methods. The values are defined as follows:-

- Block - As the mouse moves our hot areas a semi-transparent block of color indicates the drop area.
- Square - Visual Studio 2005 style square indicators are shown.
- Rounded - Visual Studio 2008 style rounded indicators are shown.

DocumentCursor
If you define the *StateCommon.Feedback* property to be *Block* then this property is used to determine if the cursor should be updated to show a document. When over a drop target a document cursor is used and when not over a hot area a document with cross is provided. You should not alter this property whilst dragging is occuring.

IsDragging
A 'get' only property used to discover if a drag operation is currently occuring.

Methods

DragStart
You should call this when you want a drag operation to be started. The first parameter is the screen position of the mouse when the drag started. A second parameter of type *PageDragEndData* is used to specify the set of pages that are being dragged. It also contains a reference to the control that is initiating the operation and allows the targets to make decisions about the eligibility of the drag based of the source and set of pages involved.

DragMove
Each time the mouse moves during the drag operation you should call this method in order to have the visual feedback updated. It takes just a single parameter which is the new screen point of the cursor. The targets are tested each time to determine which now best matches the updated mouse position.

DragEnd
When dragging has finished with success this method is called so that any appropriate target can be invoked. You need to provide the new screen point of the cursor.

DragQuit
To cancel the current dragging operation call this method without any parameters.

NOTE: These methods are all virtual and so you can customize the implementation by deriving from the *DragManager* and then overriding the methods. You might want to do this if you have special logic you want to apply in the *DragStart* that decides if the dragging is allowed to continue based on the actual pages being dragged.

Drag Enabling Controls

To enable page dragging from your control you need to add calls to the *IDragPageNotify* interface. If you require your control to be a page dragging target then you need to expose and implement the *IDragTargetProvider* interface. Both interface needs to be handled if you require dragging from as well as dropping onto the same control. A sample of how to achieve both of these is shown by deriving a class from *TreeView* and allowing individual tree nodes to be dragged away as well as new nodes added when pages are dropped onto it. You can find the full source code in the *Advanced Page Drag + Drop* sample that is contained in the *Krypton Workspace Examples* solution that comes with *Krypton*. You can quickly find this solution by running the *Krypton Explorer* and selecting the *Resources* tab.

Implementing IDragPageNotify

Derive class from TreeView
Derive a new class from *TreeView* called *PageDragTreeView* that has the following simple definition.

```
/// <summary>
/// TreeView customized to work with KryptonPage drag and drop.
/// </summary>
public class PageDragTreeView : TreeView
{
    /// <summary>
    /// Initialize a new instance of the PageDragTreeView class.
    /// </summary>
    public PageDragTreeView()       {
    }
}
```

Store a IDragPageNotify reference using a property
Add the following private field and exposed property so that a developer can assign a *IDragPageNotify* instance to the control. Notice that we give the exposed property a *Browsable(false)* setting as we do not want the property exposed in the properties window at design time. Another property called *DesignerSerializationVisiblity* is used to prevent any generated code being produced for the property when the form it is contained inside is saved.

```
private IDragPageNotify _dragPageNotify;

/// <summary>
/// Gets and sets the interface for receiving page drag notifications.
/// </summary>
[Browsable(false)]
[DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)]
public IDragPageNotify DragPageNotify    {
    get { return _dragPageNotify; }
    set { _dragPageNotify = value; }
}
```

Add Node drag support
We need to intercept the mouse down, mouse move and mouse up events so that we can detect and process the user trying to drag an individual tree node. To start we need some extra private fields that are used to cache dragging information during the dragging process. A boolean *_dragging* is used to remember if we are currently dragging or not. The *_dragRect* and *_dragNode* are updated when the mouse is pressed down to remember the rectangle the mouse has to move outside of before the drag starts, along with the node that the user has pressed down on. Finally the _dragPage is a temporary *KryptonPage* instance created for the duration of the drag operation.

The *OnMouseDown* code is simple and looks to see when the left mouse button is pressed as the hint that dragging might start when the mouse moves.

```
/// <summary>
/// Raises the MouseDown event.
/// </summary>
/// <param name="e">A MouseEventArgs that contains the event data.</param>
protected override void OnMouseDown(MouseEventArgs e)
{
    // Grab the node under the mouse
    Point pt = new Point(e.X, e.Y);
    TreeNode nodeDown = GetNodeAt(pt);

    // Try and ensure the node is selected on the mouse down
```

```
    if ((nodeDown != null) && (SelectedNode != nodeDown))
      SelectedNode = nodeDown;

    // Mouse down could be a prelude to a drag operation
    if (e.Button == MouseButtons.Left)
    {
      // Remember the node as a potential drag node
      _dragNode = nodeDown;

      // Create the rectangle that moving outside causes a drag operation
      _dragRect = new Rectangle(pt, Size.Empty);
      _dragRect.Inflate(SystemInformation.DragSize);
    }
    base.OnMouseDown(e);
  }
```

The *OnMouseMove* is responsible for deciding if dragging can be started and if already started if it should generate a drag move. The implementation of the actual *PageDragMove* and *PageDragStart* will be shown in a moment.

```
  /// <summary>
  /// Raises the MouseMove event.
  /// </summary>
  /// <param name="e">A MouseEventArgs that contains the event data.</param>
  protected override void OnMouseMove(MouseEventArgs e)    {
    Point pt = new Point(e.X, e.Y);

    // Are we monitoring for drag operations?
    if (_dragNode != null)       {
      // If currently dragging
      if (Capture && _dragging)             PageDragMove(pt);
      else if (!_dragRect.Contains(pt))             PageDragStart(pt);
    }
    base.OnMouseMove(e);
  }
```

The *OnMouseUp* always ends the dragging operation if it is taking place. Depending on if the left mouse is released or not determines if the drag is completed with success or aborted. Implementation of the actual *PageDragEnd* and *PageDragQuit* will be shown in a moment.

```
  /// <summary>
  /// Raises the MouseUp event.
  /// </summary>
  /// <param name="e">A MouseEventArgs that contains the event data.</param>
  protected override void OnMouseUp(MouseEventArgs e)    {
    if (_dragging)       {
      if (e.Button == MouseButtons.Left)
        PageDragEnd(new Point(e.X, e.Y));
      else
        PageDragQuit();
    }

    // Any released mouse means we have ended drag monitoring
    _dragNode = null;

    base.OnMouseUp(e);
  }
```

So far the code has been concerned with the specifics of detecting and operating a drag in the context of the *TreeView*control. Your own version of this code will vary depending on what is relevant for your own control. Something like a *Button* control would be simpler as you do not need to remember which node the drag is related to. Finally we have the actual implementation of the four helper methods that the above code calls into. The most complex of these is the *PageDragStart*.

```
  private void PageDragStart(Point pt)    {
    if (DragPageNotify != null)       {
      // Create a page that will be dragged
      _dragPage = new KryptonPage();          _dragPage.Text = _dragNode.Text;

      // Give the notify interface a chance to reject the attempt to drag
      PageDragCancelEventArgs de = new PageDragCancelEventArgs(PointToScreen(pt), new KryptonPage[] { _dragPage });
```

```
            DragPageNotify.PageDragStart(this, null, de);
            if (de.Cancel)
            {
                // No longer need the temporary drag page
                _dragPage.Dispose();
                _dragPage = null;
            }
            else
            {
                _dragging = true;
                Capture = true;
            }
        }
    }
```

PageDragStart creates a new *KryptonPage* instance that will passed into the dragging operation. Your own implementation would need to populate the page fields in a way appropriate for your application but in this example we only set the *Text* property of the page to match the *Node* text. It then calls into the *IDragPageNotify.PageDragStart* method and only if the method has not been cancelled does the drag then *Capture* the mouse to guarantee mouse input until the operation ends. If the drag start is cancelled it disposes of the temporary page.

The *PageDragMove routine* is very simple and just informs the *IDragPageNotify* interface of the new screen location of the mouse.

```
    private void PageDragMove(Point pt)
    {
        if (DragPageNotify != null)
            DragPageNotify.PageDragMove(this, new PointEventArgs(PointToScreen(pt)));
    }
```

Implementation of the *PageDragEnd* and *PageDragQuit* are almost identical. The only difference is the *PageDragQuit* disposes of the temporary page because the drag failed and so the page was not transferred to the drop target, whereas the *PageDragEnd* does not dispose of the page but does remove the *Node* associated with the drag. Your own control might not want to remove the *Node* that caused the drag because you want it to be dragged over and over again for creating multiple new pages.

```
    private void PageDragEnd(Point pt)
    {
        if (DragPageNotify != null)
        {
            // Let the target transfer the page across
            DragPageNotify.PageDragEnd(this, new PointEventArgs(PointToScreen(pt)));

            // Remove the node that caused the transfer
            Nodes.Remove(_dragNode);

            // Transferred the page to the target, so do not dispose it
            _dragPage = null;

            // No longer dragging
            _dragging = false;
            Capture = false;
        }
    }

    private void PageDragQuit()
    {
        if (DragPageNotify != null)
        {
            DragPageNotify.PageDragQuit(this);
            // Did not transfer the page to the target, so dispose it
            _dragPage.Dispose();
            _dragPage = null;

            // No longer dragging

            _dragging = false;
            Capture = false;
        }
    }
```

You now have a *TreeView*that can be used to drag nodes so they become pages inside the *Navigator*, *Workspace* or other compatible controls.

Implementing IDragTargetProvider

Add the IDragTargetProvider to the class definition

We are going to add to the previous class we created above by adding the ability to drop pages onto the *TreeView* in order to add extra *Node* instances at the root level. To begin we need to modify the class so that it exposes the *IDragTargetProvider* interface.

```
public class PageDragTreeView : TreeView, IDragTargetProvider
```

Implement the GenerateDragTargets method

Our new interface has just a single method called *GenerateDragTargets* that returns a list of drag targets for the control. In our case we are going to return just a single target that represents the entire control client area. We need to create a drag target class that knows how to take a *KryptonPage* and process it for our control which we will do by simply creating a new *Node* and adding it to the end of the root nodes collection. Here is the trivial implementation of the *GenerateDragTargets*.

```
/// <summary>
/// Generate a list of drag targets that are relevant to the provided end data.
/// </summary>
/// <param name="dragEndData">Pages data being dragged.</param>
/// <returns>List of drag targets.</returns>
public DragTargetList GenerateDragTargets(PageDragEndData dragEndData)    {
  DragTargetList targets = new DragTargetList();
  targets.Add(new DragTargetTreeViewTransfer(RectangleToScreen(ClientRectangle), this));
  return targets;
}
```

If you wanted to provide different actions depending on where in the control the drop occurs then you would create multiple drag targets and return the whole set from this method. For example you might want to create a drop target per visible node so that the drop action is to add the page as a new child of that particular node.

Inherit a class from DragTarget

All drag target implementations must derive from the base class *DragTarget*. Our class will take and store an incoming reference to the owning instance so that we can access the control during the processing of the drag methods that we also need to implement. Here is the beginning of our class.

```
public class DragTargetTreeViewTransfer : DragTarget
{
  private PageDragTreeView _treeView;

  /// <summary>
  /// Initialize a new instance of the DragTargetTreeViewTransfer class.
  /// </summary>
  /// <param name="rect">Rectangle for screen/hot/draw areas.</param>
  /// <param name="navigator">Control instance for drop.</param>
  public DragTargetTreeViewTransfer(Rectangle rect, PageDragTreeView treeView)
    : base(rect, rect, rect, DragTargetHint.Transfer)
  {
    _treeView = treeView;
  }
```

Notice that the base constructor takes three rectangles and a drag hint value. If creating your own drag targets you will need to understand the relationship between these three rectangles and how the hint is also used to enable useful visual feedback. The three rectangles are:-

- **Screen Rect**
  Rectangle of the entire control area in screen coordinates. If you add several targets for the same control then this rectangle should be same value for all of those targets. This allows the drag manager to recognize that the set of targets are all related and allows the drag manager to aggregate them together when providing visual feedback to the user. For example the *Square* and *Rounded* feedback settings often show a graphic with left/right/top/bottom indicators all grouped together. It can do this because this property is the same for all the left/right/top/bottom targets and so it knows it can aggregate them together visually.
- **Hot Rect**
  Screen coordinates of the rectangle that should activate the target. So when the user moves the mouse over this rectangle the drag manager can then highlight the screen to show that this target is the current one. Note that this rectangle is only used for the *Block* feedback setting and ignored otherwise. The *Square* and *Rounded* feedback settings calculate their own hot rectangles instead.
- **Draw Rect**
  This is the rectangle in screen coordinates that is highlighted on the screen when the target becomes the current target.

The final base constructor parameter is a *DragTargetHint* enumeration value and is used to inform the feedback drawing about the type of operation this target will perform. In our case we provide the *Transfer* value because we will transfer the source pages into ourself. This hint allows the visual feedback to show an appropriate graphic to the user.

Implement IsMatch

As the mouse moves around the screen each target is asked if the new mouse position is a match for the target. The first target that returns *True* will become the current target for a drop. By default the base class implementation compares the incoming point against the hot rectangle that was provided in the constructor. In our case we want to add a little extra logic to prevent the dragging of a *Node* from ourself being dropped back onto us again. Here is the code that rejects the drop onto the same instance of our *PageDragTreeView* as the source of the drag.

```
/// <summary>
/// Is this target a match for the provided screen position.
/// </summary>
/// <param name="screenPt">Position in screen coordinates.</param>
/// <param name="dragEndData">Data to be dropped at destination.</param>
/// <returns>True if a match; otherwise false.</returns>
public override bool IsMatch(Point screenPt, PageDragEndData dragEndData)
{
  // Cannot drag back to ourself
  if ((dragEndData.Source != null) &&
      (dragEndData.Source is PageDragTreeView) &&
      (dragEndData.Source == _treeView))
    return false;
  else
    return base.IsMatch(screenPt, dragEndData);
}
```

The *PageDragEndData* parameter contains information about the dragged pages as well as a reference to the source control of the drag operation.

Implement PerformDrop

Finally we need to implement the action to take when the drop occurs on our target. We just enumerate over all the pages being dragged and create a new *Node* for each one that is then added to the end of the root nodes collection. As a final action the last node is selected.

```
/// <summary>
/// Perform the drop action associated with the target.
/// </summary>
/// <param name="screenPt">Position in screen coordinates.</param>
/// <param name="data">Data to pass to the target to process drop.</param>
/// <returns>True if the drop was performed and the source can remove any pages.</returns>
public override bool PerformDrop(Point screenPt, PageDragEndData data)
{
  // Create a node for each page
  foreach (KryptonPage page in data.Pages)
  {
    // Create node and populate with page details
    TreeNode node = new TreeNode();
    node.Text = page.Text;


    // Add to end of root nodes
    _treeView.Nodes.Add(node);
  }

  // Take focus and select the last node added
  if (_treeView.Nodes.Count > 0)      {
    _treeView.SelectedNode = _treeView.Nodes[_treeView.Nodes.Count - 1];          _treeView.Select();
  }
  return true;
}
```

You now have a fully functional *TreeView* that can interact with other *KryptonPage* dragging source and targets.

Docking

Learn more about the capabilities of *Krypton Docking* using the following sections.

Introduction
- Getting Started
- Page Creation

Intermediate
- Flags
- Persistence
- Hierarchy

Events
- Persistence
- User Requests
- Controls
- Drag & Drop

Docking - Getting Started

How To Add Docking To Your Application

This tutorial is going to explain how the docking system works by walking you through the process of adding docking capabilities to a new application. At the end of the process you should have a basic understanding of how the docking system works. This will allow you to then pick and choose from the remaining docking topics depending on your application needs. Ideally we recommend that you scan all the intermediate topics in order to get a feel for the capabilities available from the docking system.

Finished Example

At the end of the tutorial you should have Figure 1 as the end result. This end result will show how to add docking to a *KryptonPanel* instance. The panel has a filler control that is an instance of the *KryptonDockableWorkspace* that holds the pages '*Page 1*' and '*Page 2*' as seen in Figure 1 below. This is the filler control that takes up all the remaining space after the docked/auto hidden pages are sized and positioned. At the bottom of the panel are two docked pages named '*Page 3*' and '*Page 4*', on the left edge is an auto hidden group with three pages named '*Page 5*', '*Page 6*' and '*Page 7*' and finally a single floating window is displayed to the bottom right of Figure 1 containing two more pages titled '*Page 8*' and '*Page 9*'.

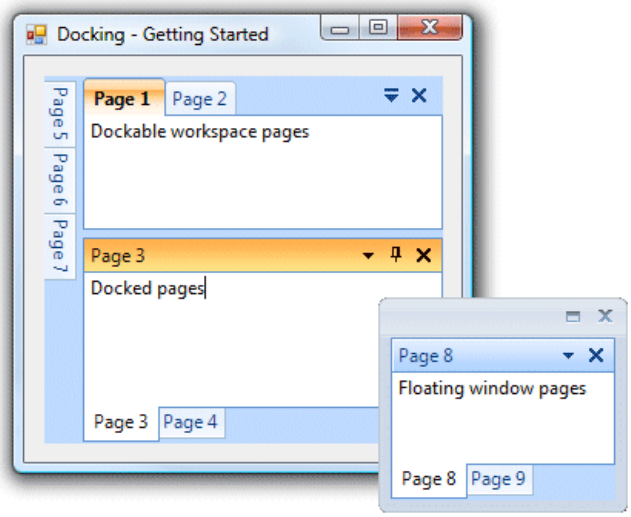


Figure 1 - Tutorial end result

Design Time Actions

Begin by creating a new windows forms project that results in an empty *Form* instance. Drag and drop a *KryptonPanel* onto the form and position it to take up all the client area except a small border around the edge. Now drag and drop a *KryptonManager* instance onto the *Form* followed by dropping a *KryptonDockingManager* instance. Both the *KryptonManager* and *KryptonDockingManager* will be added to the icon tray area of the design surface. The result of these actions can be seen in Figure 2.

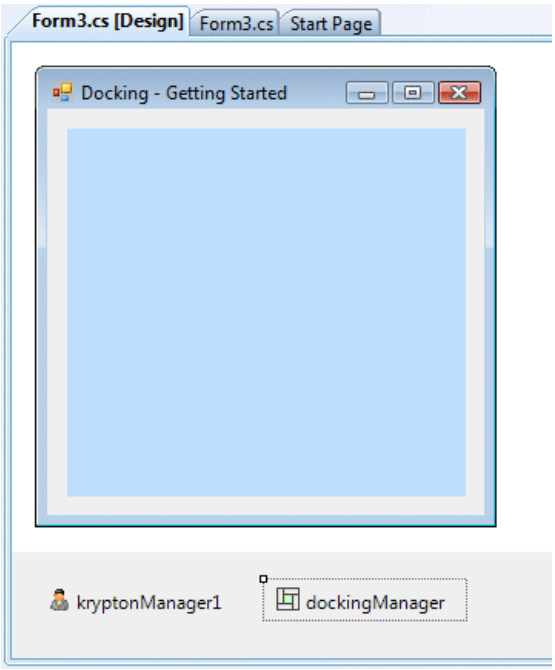


Figure 2 - KrytonPanel and icon tray components

Our final design time task is to setup the panel filler control. Drag a *KryptonDockableWorkspace* from the *Toolbox* and drop it onto the *KryptonPanel*. Once dropped you should use the properties window to set the *Dock* property of the *KryptonDockableWorkspace* to be *Fill*. This ensures that the dockable workspace takes up all the space within the panel. Later on when we add docking elements the dockable workspace will automatically be reduced to the space that is left over after positioning the docked and auto hidden elements. Without setting the *Dock* property we would have docking elements overlapping the dockable workspace leading to user confusion.

In your own application you could ignore the dockable workspace and instead place whatever is appropriate for your own needs. In that scenario just drop your own control into the panel but remember to set the *Dock* property to *Fill*. Alternatively you might not want any control to act as the panel filler in which case ignore this step entirely. The docking system is perfectly capable of working when there is no filler control taking up the remainder space.

Referencing Krypton Assemblies

Dragging and dropping the various *Krypton* components and controls above will have added project level references to the appropriate *Krypton* assemblies. You can add references manually by right clicking the *References* folder in your project and then using the '*Add Reference...*' context menu option. The presented dialog box is then used to add references by selecting the *.NET* tab and then searching the presented list of assemblies for the *ComponentFactory.Krypton* assemblies. Alternatively select the *Browse* tab and then navigate to the file location of the assemblies.

Once the project references are in place we need to add some '*using*' statements to the forms code file. This makes it easier to write code that involves the *Krypton* components without the need to fully scope them. Open up the form code file and add the following statements to the existing entries...

```
using ComponentFactory.Krypton.Toolkit;
using ComponentFactory.Krypton.Navigator;
using ComponentFactory.Krypton.Workspace;
using ComponentFactory.Krypton.Docking;
```

Creating Pages

Now we need to add some simple code that creates *KryptonPage* instances that can be added as docking pages for display. Our example code merely creates pages that contain a simple *KryptonRichTextBox* instance that is docked to fill the entire page client area. Your own real world applications would create something more interesting. For more details on the best practice for creating docking pages see the Docking Page Creation section.

Take careful note of the line setting the '*page.UniqueName*' property. It is important that each page in your docking system has a unique name because the docking system always assumes that all pages have a unique value for this property. Although each *KryptonPage* when constructed is given a unique name by generating a *GUID* you are advised to set this yourself. Using your own names such as '*Properties*', '*Output*' and '*Document*' is much easier than working with a *GUID* like '483CC32A643F4AB7483CC32A643F4AB7'.

Copy the following code into your form code file for use later on.

```
private int _count = 1;
private KryptonPage NewPage()    {
    // Create and uniquely name the page
    KryptonPage page = new KryptonPage();
    page.Text = "Page " + (_count++).ToString();
    page.TextTitle = page.Text;
    page.UniqueName = page.Text;

    // Add rich text box as content of the page
    KryptonRichTextBox rtb = new KryptonRichTextBox();
    rtb.StateCommon.Border.Draw = InheritBool.False;
    rtb.Dock = DockStyle.Fill;
    rtb.Text = "Page Content";
    page.Controls.Add(rtb);

    return page;
}
```

Adding Dock capabilities

We are now ready to actually add some docking specific code. At design time you need to double click the caption area of the form in order to auto generate an event handler for the *Form.Load* event. You should always wait until the load event occurs before executing any docking code. Do not perform docking actions in the constructor as the form has not been fully created and this will cause problems to occur. We only need three lines of code to setup the docking capabilities in this example. Add the following lines to the load event...

```
private void Form_Load(object sender, EventArgs e)

{
    KryptonDockingWorkspace w = dockingManager.ManageWorkspace("Workspace", kryptonDockableWorkspace1);
    dockingManager.ManageControl("Control", kryptonPanel1, w);
    dockingManager.ManageFloating("Floating", this);
```

All three methods begin with '*Manage*' and this indicates that they add docking capabilities based on the passed in *Control* or *Form* instance. Our first method is *ManageWorkspace* and is used to add an existing *KryptonDockableWorkspace* into the docking system. Note that you cannot add a standard *KryptonWorkspace* into the docking system, only the derived class *KryptonDockableWorkspace* that is a workspace that has been customized to work in conjunction with the docking system. The first parameter of the method call is a string that provides a unique name for the docking element that manages the dockable workspace. In this case we use the name '*Workspace*'. We need to name the element so that we can refer to the element later on when using other *KryptonDockingManager* methods. It is also possible to add more than one dockable workspace to the docking system and so giving them each a different name allows us to correctly refer to the one of interest. The second parameter of the *ManageWorkspace* method is a reference to the existing dockable workspace control, which we added at design time in a previous step.

Line two makes a call to *ManageControl* and adds docking capabilities to an existing *Control* derived reference. It adds the ability to have docked and auto hidden pages and so we have passed in the *'kryptonPanel1'* reference. This then allows us to have docked pages against the edges of the panel and auto hidden groups at the panel edges. We have provided a name for this docking element of '*Control*' so we can refer to the docking element in the future. The final parameter is a reference to the filler control. In our case this is the *KryptonDockingWorkspace* element created on the line before. We need to pass this into the method so that the docking indicators are correctly provided on the filler area of the control.

Our third line calls *ManageFloating* and as the name suggests it adds the ability to have floating windows in the docking system. As always we have given it a name so we can refer to it later on, in this case '*Floating*'. The second parameter is a reference to the *Form* that makes up the main application window. This form reference is used by the docking element as the parent of the docking windows that are created. This ensure the floating windows are associated with the main application form and so are automatically minimized when the main form is minimized and also ensures they always appear above the main form.

These three methods are all you need to setup the majority of docking scenarios.

Creating Initial Pages

Now we have docking capabilities setup we want to actually add some pages. In our tutorial we will add all the pages inside the *Form.Load* event but you could of course add your own pages in response to user actions such as pressing buttons or the loading of files. Add the following four lines to create the pages seen in configuration seen in Figure 1...

```
    dockingManager.AddToWorkspace("Workspace", new KryptonPage[] { NewPage(), NewPage() });
    dockingManager.AddDockspace("Control", DockingEdge.Bottom, new KryptonPage[] { NewPage(), NewPage() });
```

```
    dockingManager.AddAutoHiddenGroup("Control", DockingEdge.Left, new KryptonPage[] { NewPage(), NewPage(), NewPage() });
    dockingManager.AddFloatingWindow("Floating", new KryptonPage[] { NewPage(), NewPage() });
}
```

All four methods begin with '*Add*' and this indicates that they are adding pages into the relevant docking elements. Our first method is slightly different from the others as it starts with '*AddTo*' which indicates that it is adding pages into an existing control rather than creating any new controls or docking elements. The *AddToWorkspace* method is used to add pages into a dockable workspace that has already been made part of the docking system via a *ManageWorkspace* call. We provide '*Workspace*' as the first parameter as that was the name used in the original *ManageWorkspace* call. If we had added several different dockable workspace instances then this name would allow us to identity the one of interest. The second parameter is an array of *KrytonPage* references that are to be added into the workspace. To make the code easy to read we just make multiple calls to our *NewPage()* helper method as added in an earlier step. You can see the added pages as '*Page 1'* and '*Page 2'* in Figure 1.

The second method call *AddDockspace* is being used to add two pages to the bottom edge of the panel. We identify the target control using the first parameter and use the second parameter to specify which of the four control edges we are adding against. Finally our third parameter is the array of pages to be added. We provide two pages and so the docking area will have two pages in a tabbed appearance. You can see them as '*Page 3'* and '*Page 4'* in Figure 1. Adding a new auto hidden group is just as simple, just call *AddAutoHiddenGroup* and provide the same set of parameters as the previous *AddDockspace* method. You can see the auto hidden group on the left edge of the panel in Figure 1.

Finally line four adds a floating window that shows two pages in tabbed appearance. Just call *AddFloatingWindow* and provide the name of element that was just in the *ManageFloating* call along with an array of pages to be added. You can see the created floating window at the bottom right corner of Figure 1.

You can now compile and run the application.

Panel Filler Alternatives

If you prefer to use a navigator instead of the workspace as the panel filler you can replace the *KryptonDockableWorkspace* instance with the *KryptonDockableNavigator* control instead. This is a version of the navigator that is customized to integrate into the docking system. In this case you would simply replace the first two lines of the above *Form.Load* event code with the following similar code...

```
KryptonDockingNavigator n = dockingManager.ManageNavigator("Navigator", kryptonDockableNavigator1);
dockingManager.ManageControl("Control", kryptonPanel1, n);
```

Then later on you would use the following *AddToNavigator* instead of the *AddToWorkspace*...

```
dockingManager.AddToWorkspace("Navigator", new KryptonPage[] { NewPage(), NewPage() });
```

Note that you can even omit the use of a navigator or workspace in which case you would remove the first line of *Form.Load* event code and remove the third parameter from the *ManageControl* call. You could leave the filler area of the panel empty or add some other unrelated control that does not become part of the docking hierarchy. The docking system can work perfectly well without any filler control or with a filler control that is not a navigator or workspace.

Docking Page Creation

Best Practice
Once you have your docking system up and working your focus will shift to creating the pages that will be displayed. This quick tutorial shows our recommended best practice approach that makes it easy for you to design and then use pages in the docking system. In fact the same approach could be used for creating pages that are intended for use in the *KryptonNavigator* or *KryptonWorkspace*.

Step 1 - Create a User Control
We begin by creating a new user control. Right click your project entry in the solution explorer and select '*Add*' followed by '*New Item...*' in order to bring up the '*Add New Item*' dialog box that presents different items for creation. Double click the '*User Control*' option as can be seen in Figure 1. This will result in a new user control derived class being created and the designer for the control should be opened by default.

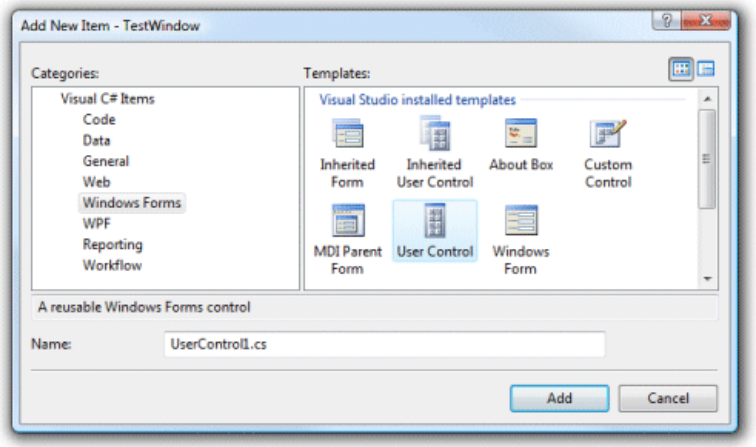


Figure 1 - Add New Item Dialog

Step 2 - Design your User Control
This is where you need to use the control designer to build the control content that is appropriate for your application. That might be a entry form with many input controls or maybe a charting or reporting control. To keep this tutorial simple we add a single *KryptonButton* by dragging it from the Toolbox and dropping it on the design surface. This results in the following simple appearance as seen in Figure 2.

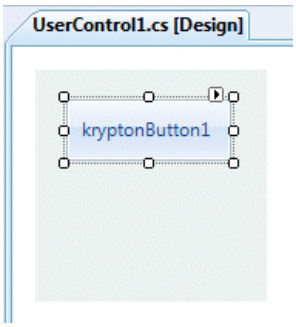


Figure 2 - Designed UserControl

Step 3 - Create a Class
To make this user control usable in the docking system we need to add a new class. Right click your project entry in the solution explorer and select '*Add*' followed by '*New Item...*' in order to bring up the '*Add New Item*' dialog box that presents different items for creation. This click double click the '*Class*' option. This will result in a simple class outline being created as can be seen in Figure 3.

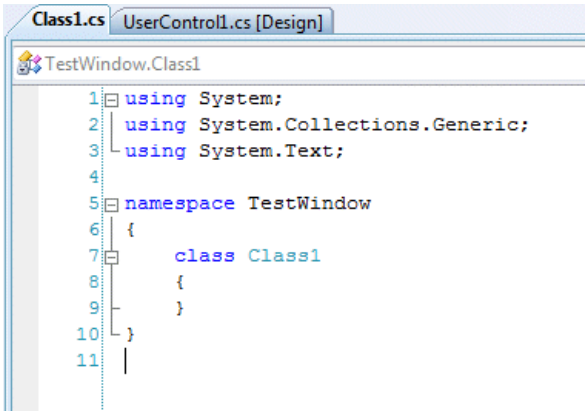


Figure 3 - NewClass file

Step 4 - Add 'using' Statement

Place an additional 'using' statement at the top of the file so we can refer to the *KryptonPage* class.

```
using ComponentFactory.Krypton.Navigator;
```

Step 5 - Inherit from 'KryptonPage'

Replace the class definition line so that we create a public class that inherits from the *KrytonPage*.

```
public class Class1 : KryptonPage
```

Step 6 - Create UserControl in Constructor

We need to add a constructor that creates an instance of the user control we created earlier and then adds it to the set of child controls for the page. As part of the process each set the *Dock* property of the user control to be *DockStyle.Fill* so that the user control is automatically size to take up the entire client area of the page. If you do not need this action then simply leave out the line that sets this property. Our finished code is very compact and looks like the following...

```
using System;
using System.Collections.Generic;
using System.Text;
using ComponentFactory.Krypton.Navigator;

namespace TestWindow
{
    public class Class1 : KryptonPage
    {
        public Class1()
        {
            UserControl1 content = new UserControl1();
            content.Dock = System.Windows.Forms.DockStyle.Fill;
            Controls.Add(content);
        }

    }
}
```

Step 7 - Use the custom page

Using our custom page is now trivial. Here is an example of adding the page docked against the left edge of a target control.

```
dockingManager.AddDockspace("Control", DockingEdge.Left, new KryptonPage[] { new Class1() });
```

The advantage of this approach is that you can now modify the contents of the page by double clicking the user control class and using the designer to drag and drop controls as needed. In practice you would probably want to use more appropriate names for the user control and page class rather than the defaults. For example '*UserControlProperties*' in place of '*UserControl1'* and '*PropertiesPage*' in place of '*Class1*' would be more readable if creating a docking page for showing a set of property values.

Docking Flags

Controlling User Actions

Each *KryptonPage* has a set of flags that are used to restrict page level user actions within the docking system.

To view the page level flags you should select the page at design time so that the page properties are displayed in the properties pane. At the bottom of the properties pane you will see a verb called '*Edit Flags*' as shown in Figure 1. On pressing the link you will be presented with a dialog box showing the full set of page flags including those applicable to the docking system. Figure 2 shows an example of the dialog.

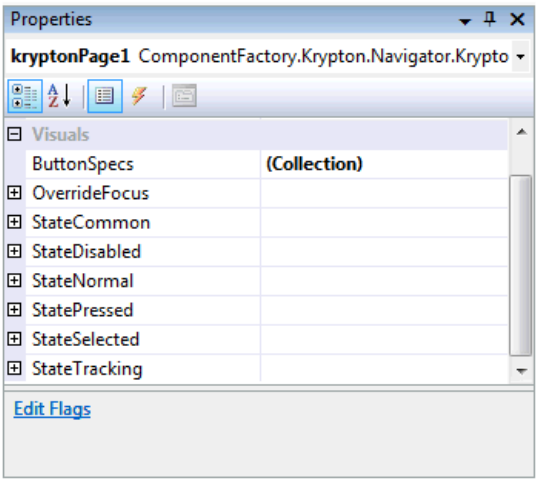


Figure 1 - KryptonPage 'Edit Flags' verb

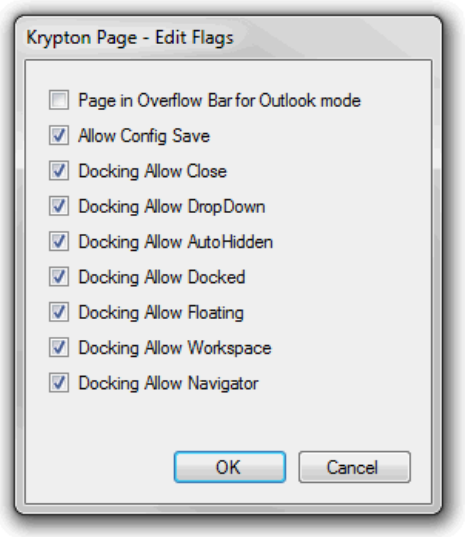


Figure 2 - KryptonPage Flags Editing

To access the flags using code you can use the *KryptonPage.Flags* property and the helper methods *KryptonPage.ClearFlags*, *KryptonPage.SetFlags* and *KryptonPage.AreFlagsSet.*

DockingAllowClose

This flag causes a close button to appear on the page header. When the flag is cleared the header button is removed and the user prevented from closing the flag. This flag also influences the docking context menu by disabling the hide option when the flag is not defined. You can see the docking context menu three different ways. You can right click the page tab, right click the page header or left click the drop down button that appears on the page header. Note that this flag only affects user interaction and you can still close the page programmatically at any time.

DockingAllowDropDown
Use this flag to determine if the drop down button appears on the docking caption area. By default the flag is defined and so downward pointing arrow is shown of the docked and floating windows so that a drop down menu can be displayed. Clearing this flag prevents that drop down arrow button from showing. It does not affect the ability to right click the header or right click the tabs in order to show the same context menu as appears when pressing the drop down button.

DockingAllowDocked
When the user performs a drag and drop operation this flag is used to decide if the page is allowed to be dropped into the docked state. Clear this flag if you wish to prevent the user from dropping into a docked state. When the page is in the auto hidden state and this flag is defined as true an extra unpin button is placed on the page header so that he user can switch the page from auto hidden to docked. The docking context menu has an option for changing the page to docked and when this flag is cleared that menu option is disabled. Note that this flag only affects user interaction and you can still programmatically make the page docked.

DockingAllowAutoHidden
When defined an extra pin button is placed onto the page header of a docked page that allows the user to switch the page to auto hidden. Also affected is the docking context menu that has an option for switching a page into the auto hidden state. When this flag is cleared the menu option is disabled. Note that this flag only affects user interaction and you can still programmatically make the page auto hidden.

DockingAllowFloating Dragging a page will by default cause that page to become floating when this flag is defined. Clear this flag to prevent a page becoming floating at the start of the drag operation. The docking context menu has an option for changing the state of a page to floating and when this flag is cleared that menu option is disabled. Note that this flag only affects user interaction and you can still programmatically make the page floating.

DockingAllowWorkspace
This flag is only used when you have added a *KryptonDockableWorkspace* into the docking hierarchy. In that scenario this flag allows a page to be dragged into the workspace area. Clear this flag to prevent the user from dragging, or using the docking context menu, to transfer a page into the workspace area. Note that this flag only affects user interaction and you can still programmatically make the page appear in the workspace.

DockingAllowNavigator
This flag is only used when you have added a *KryptonDockableNavigator* into the docking hierarchy. In that scenario this flag allows a page to be dragged into the navigator control. Clear this flag to prevent the user from dragging, or using the docking context menu, to transfer a page into the workspace area. Note that this flag only affects user interaction and you can still programmatically make the page appear in the navigator.

Docking Persistence

Applicable Methods: SaveConfigToArray, LoadConfigFromArray
SaveConfigToFile, LoadConfigFromFile SaveConfigToXml, LoadConfigFromXml
SaveConfigToStream, LoadConfigFromStream

Persistence Formats
You can save and load the docking configuration in a variety of different formats in order to suit the needs of your application. Use the *SaveConfigToFile* and *LoadConfigFromFile* methods in order to persist to files in the XML format. This is useful if you need to retain a configuration when an application exits in order to restore it again when next run. Alternatively you can use the *SaveConfigToArray* and *LoadConfigFromArray* pair of methods that persist as an array of bytes. This makes it easy to place the data into a database, transfer it over a network connection or just store it within your application. For greater control you can use the *SaveConfigToXml* and *LoadConfigFromXml* pair that expect *XmlTextWriter* and *XmlTextReader* instances. Finally the last pair *SaveConfigToStream* and *LoadConfigFromStream* provide maximum flexibility because they accept a generic stream for the storage. This allows you to easily integrate the configuration information into your own persistence mechanism.

Information Stored
The dynamic contents of the docking hierarchy are saved into the configuration but the static elements are not. So if your docking hierarchy has a floating capability at save time but that docking element is not present at the reload then the floating ability is not recreated. This is because the *KryptonDockingFloating* is a static element that manages a capability and is only ever created by the programmer. Floating windows are dynamic content and details of each floating window are saved into the configuration. On reload the floating windows will be recreated as long an appropriate *KryptonDockingFloating* is present that can contain the recreated floating window definitions.

Only basic information about each page are persisted. The *UnqiueName* of the page and the *Visible* state are saved but no other page details. This is because at load time an existing page with a matching *UniqueName* is expected to be found and then positioned at the location indicated by the loading state. If the page does not exist then you need to hook into the *RecreateLoadingPage* event and create the page so it can be positioned.

Global Custom Data
You may need to store additional application specific data along with the configuration for use when reloading. You can do this quite easily by hooking into the *GlobalSaving* and *GlobalLoading* events. The saving event will provide an *XmlWriter* reference that should be used to save your extra information. Create additional XML elements with whatever information you need to persist. Loading provides an *XmlReader* that can be used to traverse and load back that same information.

We recommend saving your own version number into the custom data so that in the future you can recognize a change in the way you have stored the data. This makes it easier to change the storage and then still be able to recognize older formats and be able to process them.

Page Custom Data
Handling per-page custom data is similar to the above global custom data method. Use the *PageSaving* and *PageLoading* events to hook into the process and use the event parameters to get a reference to the actual page that is being saved/loaded. You are given a *XmlWriter* for saving and *XmlReader* for loading the data. An extra feature of the loading event is the ability to modify the page reference provided as an event parameter and have that new page reference used instead of the instance passed into the event. This allows you to override the loading process and force the use of your own designated page. If you override the page reference with *null* then you will prevent any page being added at all. So you can dynamically decide if a loading page is desired and use *null* to reject it from being added into the docking hierarchy.

Recreating Pages
When the load process is stared the current set of pages within the docking hierarchy are added into a list. When a page is encountered in the loading configuration that list is scanned to see if an existing page matches the same *UniqueName* as the one defined in the configuration. If a match is found then the existing page reference is used and added back into the hierarchy. When there is no match it means the page needs to be recreated so it can then be added into the appropriate location of the docking hierarchy. To do this the *RecreateLoadingPage* event is fired. If the programmer does not hook into this event and provide a recreated page then the loading page is ignored.

Orphan Pages
When the load process is stared the current set of pages within the docking hierarchy are added into a list. At the end of the loading process the docking hierarchy will have been cleared and recreated with the configuration that has been loaded. Any page that is in the list but has not been loaded is called an orphan because it is no longer part of the docking hierarchy. To process these orphan pages you need to hook into the *OrphanedPages* event. You might do this to add the orphan pages back into the docking system and prevent them being lost. If you ignore the event then the orphan pages will be disposed.

Docking Hierarchy

Hierarchy of Docking Elements

The docking system is implemented as a tree of elements where specific docking functionality is provided by particular elements. To achieve your docking requirements you need to build up the hierarchy of elements to match the functionality you require. To make this process as simple as possible a set of helper methods starting with the word '*Manage*' are exposed by the docking manager. You can see them being used in the Docking Getting Started section. Understanding the different docking elements and how to build them into a tree is a great way to get a full understanding of the docking system. It will allow you to perform more complicated tasks such as setting up complex arrangements of pages.

Every docking element has a *Name* property that is used to uniquely identify the element within the parent collection of elements. Different elements can have the same name as long as they are not sibling to each other. Each element also has a read only property called *Path*. This returns a string that shows the name of all the elements starting from the root and reaching the target element. For example a path for a docked control would be something like '*DockingManager,Control,Left,Docked*'. You can can use the *KryptonDockingManager.ResolvePath* method that takes a path string and returns an *IDockingElement* reference as a way to quickly get access to the docking element of interest within the hierarchy. Although you will need to cast the returned reference to the specific type of element you are accessing.

KryptonDockingManager

The *KryptonDockingManager* component acts as the root of the docking hierarchy. Not only does it act as the top element but it also has all the methods and properties you need for standard interaction with the docking system. For more advanced operations you can navigate around the hierarchy of docking elements and then interact with those elements directly. Each element has a *Count* property that indicates the number of children that element has. To get access to a child just use an array indexer and it will return an element.

```
Console.WriteLine("No Of Children:{0}", dockingManager.Count);
IDockingElement c = dockingManager[0];
IDockingElement f = dockingManager["Floating"];
```

As you can see above, the array accessor can take an integer or the name of the element required. The return type is *IDockingElement* which is the base interface provided by all docking element classes. You would then need to cast the returned reference to the type you are expecting to be returned. So you would have something like this...

```
KryptonDockingControl c = (KryptonDockingControl)dockingManager[0];
KryptonDockingFloating f = (KryptonDockingFloating)dockingManager["Floating"];
```

Using this approach you can navigate down the hierarchy to any element. You can then interact with the element directly rather than using the general purpose methods provided by the docking manager. The rest of this section will describe all the different docking elements that are available and how they interrelated.

KryptonDockingControl

This element adds docked and auto hidden docking capabilities to a provided control. Typically the control provided will be a *KryptonPanel* but this is not a requirement as you can add any control that derives from the *Control* base class. The *KryptonDockingControl* element actually adds four child *KryptonDockingEdge* elements to itself inside the constructor with each *KryptonDockingEdge* instance responsible for one of the control edges. Each *KryptonDockingEdge* element will itself add two children to themselves. The children are *KryptonDockingEdgeDocked* for handling docked content and *KryptonDockingEdgeAutoHidden* for managing auto hidden groups. So creating a *KryptonDockingControl* instance actually results in the following tree of elements...

```
KryptonDockingManager                   (Name=DockingManager)
    KryptonDockingControl                 (Name=Control)
        KryptonDockingEdge                  (Name=Left)
            KryptonDockingEdgeDocked          (Name=Docked)
            KryptonDockingEdgeAutoHidden      (Name=AutoHidden)
        KryptonDockingEdge                  (Name=Right)
            KryptonDockingEdgeDocked          (Name=Docked)
            KryptonDockingEdgeAutoHidden      (Name=AutoHidden)
        KryptonDockingEdge                  (Name=Top)
            KryptonDockingEdgeDocked          (Name=Docked)
            KryptonDockingEdgeAutoHidden      (Name=AutoHidden)
        KryptonDockingEdge                  (Name=Bottom)
            KryptonDockingEdgeDocked          (Name=Docked)
```

```
KryptonDockingEdgeAutoHidden      (Name=AutoHidden)
```

The four children of the docking control have name property values of '*Left*', '*Right*', '*Top*' and '*Bottom*' which relate the edge they manage. Then each edge has children named '*Docked*' and '*AutoHidden*'. You can use those names to navigate to the docking element of interest when you need to interact with them directly instead of via the docking manager. The child collection of the *KryptonDockingControl* is fixed and so you cannot add or remove, the four child elements are constant. Also the *KryptonDockingEdge* is fixed so you cannot change the two children that are always present. *KryptonDockingManager* is an open collection and begins with no children but you can add and remove children as required.

You can create an instance of the *KryptonDockingControl* and have it added to the *KryptonDockingManager* using the following helper method...

```
dockingManager.ManageControl("Control", kryptonPanel1);
```

Alternatively create and add the instance directly using the following equivalent code...

```
KryptonDockingControl dockingControl = new KryptonDockingControl("Control", kryptonPanel1);
dockingManager.Add(dockingControl);
```

This is how you can navigate to the element that manages docked content on the bottom edge...

```
KryptonDockingControl control = (KryptonDockingControl)dockingManager["Control"];
KryptonDockingEdge edge = (KryptonDockingEdge)control["Bottom"];
KryptonDockingEdgeDocked docked = (KryptonDockingEdgeDocked)edge["Docked"];
```

KryptonDockingEdgeDocked

This element contains a set of child *KryptonDockingDockspace* elements each of which represents a single docking capable workspace. The '*dockspace*' is the name of a control derived from the *KryptonWorkspace* that is customized to work as docked within a control. There are four helper methods on this element that create and add new *KryptonDockingDockspace* instances...

```
AppendDockspace()
AppendDockspace(string name)
InsertDockspace(int index)
InsertDockspace(int index, string name)
```

The methods that do not take a '*name*' parameter will instead generate a GUID and use that as the element name. If you use the '*Append*' methods then the new dockspace element is added to the end of the collection and will also be the innermost dockspace for that edge. If you use the '*Insert*' methods to add a new instance at the start of the child collection then it will be displayed as the outermost dockspace for that edge. Adding a dockspace element does not actually cause anything to be displayed as it does not contain any pages by default. See the following section for details on how to add pages to the dockspace.

KryptonDockingDockspace

This element represents a docked workspace against a control edge. It will create an actual control instance and add it to the appropriate owning control. You can get access to this control reference by using the *KryptonDockingDockspace.DockspaceControl*. The *DockspaceControl* is a control derived from *KryptonWorkspace* and so if you want to perform complicated layout tasks for pages inside the workspace you would need to use this reference to get access to the control. Most developers will not need to do this and can instead use the helper methods listed here for adding pages...

```
Append(KryptonPage page)
Append(KryptonPage[] pages)
CellAppend(KryptonWorkspaceCell cell, KryptonPage page)
CellAppend(KryptonWorkspaceCell cell, KryptonPage[] pages)
CellInsert(KryptonWorkspaceCell cell, int index, KryptonPage page)
CellInsert(KryptonWorkspaceCell cell, int index, KryptonPage[] pages)
```

The first two methods can be used to add pages to the dockspace in all circumstances. If the dockspace does not currently have any contents then it will automatically add a new *KryptonWorkspaceCell* and then add the requested pages into it. If the dockspace already has a visible workspace cell then it will instead append the provided pages into that existing cell. If you already have a reference to a workspace cell instance then the four '*Cell*' methods can be used.

To add a page to the left edge of a control with a new dockspace you can use the following docking manager helper method...

```
dockingManager.AddDockspace("Control", DockingEdge.Left, new KryptonPage[] { NewPage() })
```

The same action without the helper shows how to create a dockspace and add a page to it...

```
KryptonDockingControl control = (KryptonDockingControl)dockingManager["Control"];
KryptonDockingEdge edge = (KryptonDockingEdge)control["Left"];
KryptonDockingEdgeDocked docked = (KryptonDockingEdgeDocked)edge["Docked"];
KryptonDockingDockspace dockspace = docked.AppendDockspace();     dockspace.Append(NewPage());
```

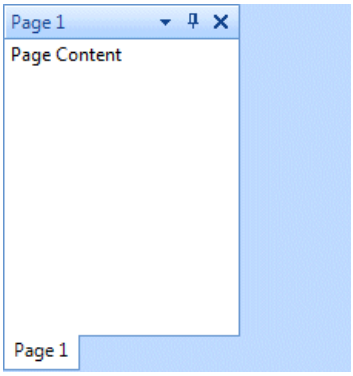


Figure 1 - Adding a left docked page

Both of the above approaches result in the Figure 1 docking layout. *Because the KryptonDockingDockspace.DockspaceControl* is a control that derives from *KryptonWorkspace* you can manipulate it to perform more complex and interesting layouts. Here we change the above code by removing the last line and replacing it with the following code in order to create three workspace cells in a vertical stack...

```
KryptonWorkspaceCell c1 = new KryptonWorkspaceCell();
KryptonWorkspaceCell c2 = new KryptonWorkspaceCell();
KryptonWorkspaceCell c3 = new KryptonWorkspaceCell();

c1.Pages.Add(NewPage());     c2.Pages.Add(NewPage());
c3.Pages.Add(NewPage());

dockspace.DockspaceControl.Root.Orientation = Orientation.Vertical;
dockspace.DockspaceControl.Root.Children.AddRange(new Component[] { c1, c2, c3 });
```

The output will be as shown in Figure 2. If you need to create complex layouts then you recommended to read the documentation Workspace Layout and then apply it to the dockspace control.

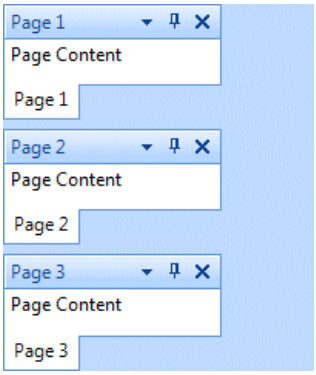


Figure 2 - Stacked cells within a single dockspace

KryptonDockingEdgeAutoHidden

The *KryptonDockingEdgeAutoHidden* element is used to manage a set of child *KryptonDockingAutoHiddenGroup* elements. Each child auto hidden group appears as a set of tab headers on the edge of the control. When first created this element contains no children and you would normally used the *KryptonDockingManager.AddAutoHiddenGroup* helper method to have a new group added. You can however navigate through the hierarchy to this element and then use one of its two helper methods to directly manipulate the collection...

```
AppendAutoHiddenGroup()
AppendAutoHiddenGroup(string name)
```

The first method does not take a '*name*' parameter and will instead generate a GUID and use that as the element name. Adding an auto hidden group element does not actually cause anything to be displayed as it does not contain any pages by default. See the following section for details on how to add pages to the group.

KryptonDockingAutoHiddenGroup

This element is very simple and only has two methods that are used to manipulate the set of pages within the auto hidden group...

```
Append(KryptonPage page)
Append(KryptonPage[] pages)
```

To add a couple of pages as an auto hidden group on the left edge of a control you can use the following docking manager helper method...

```
dockingManager.AddAutoHiddenGroup("Control", DockingEdge.Left, new KryptonPage[] { NewPage(), NewPage() })
```

The same action without the helper shows how to create an auto hidden group and add pages to it...

```
KryptonDockingControl control = (KryptonDockingControl)dockingManager["Control"];
KryptonDockingEdge edge = (KryptonDockingEdge)control["Left"];
KryptonDockingEdgeAutoHidden autoHidden = (KryptonDockingEdgeAutoHidden)edge["AutoHidden"];
KryptonDockingAutoHiddenGroup group = autoHidden.AppendAutoHiddenGroup();      group.Append(NewPage());
group.Append(NewPage());
```

Both of the above approaches result in the Figure 3 docking layout.

Figure 3 - Adding a left auto hidden group

The following hierarchy shows the result of adding the above group to the left edge.

```
KryptonDockingManager                       (Name=DockingManager)
    KryptonDockingControl                     (Name=Control)
        KryptonDockingEdge                     (Name=Left)
            KryptonDockingEdgeAutoHidden          (Name=AutoHidden)
                KryptonDockingAutoHiddenGroup      (Name=483CC32A643F4AB7483CC32A643F4AB7)
```

KryptonDockingFloating

Use this element to manage a collection of floating windows. When first created this element contains no children and you would normally used the *KryptonDockingManager.AddFloatingWindow* helper method to have a new window element added. You can however navigate through the hierarchy to this element and then use one of its two helper methods to directly manipulate the collection...

```
AddFloatingWindow()
AddFloatingWindow(string name)
```

The first method does not take a '*name*' parameter and will instead generate a *GUID* and use that as the element name. Adding a floating window element does not actually cause anything to be displayed as it does not contain any pages by default. See the following section for details on how to add pages to the floating window.

KryptonDockingFloatingWindow

This element creates a new top level window that you can access via the *KryptonDockingFloatingWindow.FloatingWindow* property. Use this property to modify the size, location or other window level properties of the floating window. The element has a single child element which is a *KryptonDockingFloatspace* that takes up the entire client area of the window and is used to store and layout the actual docking pages. You can see below the hierarchy of elements that results from adding a single floating window.

```
KryptonDockingManager                   (Name=DockingManager)
    KryptonDockingFloating                  (Name=Floating)
        KryptonDockingFloatingWindow            (Name=483CC32A643F4AB7483CC32A643F4AB7)
            KryptonDockingFloatspace                (Name=Floatspace)
```

KryptonDockingFloatspace

This element is almost identical to the *KryptonDockingDockspace* element that is described above. In this case it manages the control that is placed inside the client area of floating window. You can get access to this control reference by using the *KryptonDockingFloatspace.FloatspaceControl*. The *FloatspaceControl* is a control derived from *KryptonWorkspace* and so if you want to perform complicated layout tasks for pages inside the workspace you would need to use this reference to get access to the control. Most developers will not need to do this and can instead use the helper methods listed in the *KryptonDockingDockspace* section above.

To add a new floating window with two pages you can use the following docking manager helper method...

```
dockingManager.AddFloatingWindow("Floating", new KryptonPage[] { NewPage(), NewPage() })
```

The same action without the helper shows how to create a floating window and add two pages to it...

```
KryptonDockingFloating floating = (KryptonDockingFloating)dockingManager["Floating"];
KryptonDockingFloatingWindow window = floating.AddFloatingWindow();
window.FloatspaceElement.Append(new KryptonPage[] { NewPage(), NewPage() })
```

The output will be as shown in Figure 4.

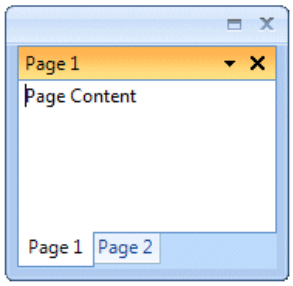


Figure 4 - Adding a floating window

KryptonDockingWorkspace

This element is different to the others in that it does not create any child elements and is used as a wrapper to manage an existing *KryptonDockableWorkspace* instance. A typical docking scenario will include a control as the filler that takes up all the remaining space when the docked and auto hidden pages have been positioned. In Visual Studio this is the control that perform document editing. When you need to same capability in the *Krypton* docking system you could add a *KryptonDockableWorkspace* control from the toolbox and drop it onto the docking control, typically a *KryptonPanel* instance. To make that control part of the docking hierarchy you need create an instance of the *KryptonDockingWorkspace* and provide it with a reference to *KryptonDockableWorkspace* that is needs to manage. This results in the user being able to drag pages from the docking pages and drop them into the dockable workspace and vice versa. Normally you would set this up using the following helper method...

```
dockingManager.ManageWorkspace("Workspace", kryptonDockableWorkspace1);
```

Alternatively you can create and add the element directly like this...

```
KryptonDockingWorkspace dockingWorkspace = new KryptonDockingWorkspace("Workspace", "Filler", kryptonDockableWorkspace1);
dockingManager.Add(dockingWorkspace);
```

KryptonDockingNavigator

This element is used to manage an existing *KryptonDockableNavigator* instance. A typical docking scenario will include a control as the filler that takes up all the remaining space when the docked and auto hidden pages have been positioned. Use this control instead of the above workspace variation when you do not need to ability to have pages dragged to be side by side and instead want a simpler traditional tab style control. To make that control part of the docking hierarchy you need create an instance of the *KryptonDockingNavigator* and provide it with a reference to *KryptonDockableNavigator* that is needs to manage. This results in the user being able to drag pages from the docking pages and drop them into the dockable naviagator and vice versa. Normally you

would set this up using the following helper method...

```
dockingManager.ManageNavigator("Navigator", kryptonDockableNavigator1);
```

Alternatively you can create and add the element directly like this...

```
KryptonDockingNavigator dockingNavigator = new KryptonDockingWorkspace("Navigator", "Filler", kryptonDockableNavigator1);
dockingManager.Add(dockingNavigator);
```

Docking Persistence Events

Persistence Events: GlobalSaving
GlobalLoading

PageSaving
PageLoading
RecreateLoadingPage
OrphanedPages

GlobalSaving
Called during the save configuration process and allows custom data to be added into the persisted data. You are provided with an *XmlWriter* reference that should be used for saving your own information. Custom data can be structured by adding new elements and attributes as needed so that the XML is structured in a logical way.

GlobalLoading
Called during the load configuration process and allows previously saved data to be reloaded. An *XmlReader* reference is provided and should be used to navigate and process the incoming information. It is recommended that you persist a version number into the custom data so that on loading you can recognize which version of your application saved the data. This makes future compatibility issues easier to handle.

PageSaving
Each time a page is saved this event is called and provided with a reference to the page along with an *XmlWriter*. Use the text writer to save any additional information you require associated with the page. For example, you might use this to save the name of a file that the page was displaying. This allows you to reload the file during loading so that the file is reconstructed with the same contents.

PageLoading
Each time a page is loaded this event is fired and provided a reference to the page along with an *XmlReader*. Load additional information using the text reader and then perform page setup actions such as creating controls for the page. You can override the page reference in order to change the page that will be added to the docking system. So you can create an entirely new page and modify the event page reference so that the new page you just created is used in place of the one provided. If you modify the event page reference to be *null* then the load process will not add any page to the docking system. This is useful if your application needs to reject the loading of individual pages.

RecreateLoadingPage
If the loading process cannot find an existing page with the same *UniqueName* as the page detailed in the configuration then this event is fired. This gives you a chance to recreate the required page. If this event is ignored, or you return *null* from the event, then the incoming page configuration is ignored and no page is added to the docking system. If you do create a page and return it from the event then the loading process continues as normal and the *PageLoading* event will then be fired for the newly created page.

OrphanedPages
At the end of the loading process there might be pages that were present in the docking system at the start of loading but are not referenced in the incoming configuration. Without hooking into this event those orphan pages are disposed and removed from the docking system. If you want to preserve some or all of those orphan pages then use this event to catch the pages and store them as appropriate.

Docking User Requests Events

User Request Events: PageCloseRequest
PageDockedRequest
PageAutoHiddenRequest PageFloatingRequest
PageNavigatorRequest PageWorkspaceRequest
ShowPageContextMenu
ShowWorkspacePageContextMenu

PageCloseRequest
This event is fired from within the *KryptonDockingManager.CloseRequest* method and so calling this method programmatically will cause the event to be fired. There are also three user actions that can occur at runtime that result in the *CloseRequest* method being called and so the *PageCloseRequest* fired. If the user presses the page close button or the user selects the hide option on the docking context menu or the user closes a floating window then a *CloseRequest* is generated. The *PageCloseRequest.CloseRequest* property can be updated by the developers event handler in order to instruct the docking system what action should be taken. Possible values are as follows...

- **None** -no action is performed and so the close request is ignored
- **RemovePage** - removes the page from the docking system but does not dispose it, so the page could be added back again later
- **RemovePageAndDispose** - removes the page and also calls Dispose so the page could not be added back again later
- **Hide** - leaves the page in the docking system but hides it from display

PageDockedRequest
Fired by two different user actions. First is using the docking context menu to request a page be switched to the docked state. Second is using the unpin button for an auto hidden group to have the group switched to docked state. Also fired if you programmatically use the *MakeDockedRequest*, *SwitchFloatingToDockedRequest* or *SwitchAutoHiddenGroupToDockedCellRequest* methods. You can cancel the event to prevent the switch from occuring.
PageAutoHiddenRequest
Fired by two different user actions. First is using the docking context menu to request a page be switched to the auto hidden state. Second is using the pin button for a docked set of pages to have the pages switched to the auto hidden state. Also fired if you programmatically use the *MakeAutoHiddenRequest* or *SwitchDockedCellToAutoHiddenGroupRequest* methods. You can cancel the event to prevent the switch from occuring.
PageFloatingRequest
Fired by two different user actions. First is using the docking context menu to request a page be switched to the floating state. Second is double clicking the header for a docked set of pages to have the pages switched to the floating state. Also fired if you programmatically use the *MakeFloatingRequest* or *SwitchDockedToFloatingWindowRequest* methods. You can cancel the event to prevent the switch from occuring.

PageWorkspaceRequest
Fired when the docking context menu is used by the user to request a page be switched to the workspace (*Tabbed Document*) state. Also fired if you programmatically use the *MakeWorkspaceRequest* method. You can cancel the event to prevent the switch from occuring.

PageNavigatorRequest
Fired when the docking context menu is used by the user to request a page be switched to the navigator (*Tabbed Document*) state. Also fired if you programmatically use the *MakeNavigatorRequest* method. You can cancel the event to prevent the switch from occuring.
ShowPageContextMenu
A docking context menu is displayed for a page in response to three different user actions. There is a drop down button on the page header that when pressed shows the docking context menu. It also appears if you right click a page tab or right click the page header. Before the context menu is displayed from any of these user actions the *ShowPageContextMenu* event is fired and allows the developer to customize the contents of the menu. You can add additional options that are specific to your own application.
ShowWorkspacePageContextMenu
When you right click a tab in a dockable workspace that is part of the docking hierarchy this event is fired. This allows you to customize the menu before it is shown to the user. You can add additional options that are specific to your own application.

Docking Controls Events

Control Resizing Events: AutoHiddenSeparatorResize
DockspaceSeparatorResize

AutoHiddenSeparatorResize
When an auto hidden slides out to be displayed there is a resize bar displayed at one edge. When you left click that resize bar this event is fired and allows you to modify the limits of the resizing operation. So you can restrict the resizing action to whatever is appropriate for your application needs.
DockspaceSeparatorResize
Each dockspace has a resize bar displayed at one edge. When you left click that resize bar this event is fired and allows you to modify the limits of the resizing operation. So you can restrict the resizing action to whatever is appropriate for your application needs.

Control Adding/Removing Events:
AutoHiddenGroupAdding AutoHiddenGroupRemoved
AutoHiddenGroupPanelAdding AutoHiddenGroupPanelRemoved
DockableNavigatorAdding DockableNavigatorRemoved
DockableWorkspaceAdding DockableWorkspaceRemoved
DockableWorkspaceCellAdding DockableWorkspaceCellRemoved
DockspaceAdding DockspaceRemoved
DockspaceCellAdding DockspaceCellRemoved
DockspaceSeparatorAdding DockspaceSeparatorRemoved
FloatspaceAdding FloatspaceRemoved
FloatspaceCellAdding FloatspaceCellRemoved
FloatingWindowAdding FloatingWindowRemoved

xxxxAdding
There are many different custom controls involved in the docking system. Each type of custom control has an event that is fired when that control type is created so that you can then customize the control if required. You might customize the control by changing the palette setting, changing state specific property values or hooking into events. If you hook into the adding event you should careful consider if you also need to hook the removing event as well. Ensure that any actions you perform are appropriately reversed in the removing event to prevent memory leaks occuring.
xxxxRemoving
There are many different custom controls involved in the docking system. Each type of custom control has an event that is fired when that control type is destroyed so that you can reverse any customizations made during the matching adding event. If you hooked into the adding event you should careful consider if you also need to hook the removing event as well. Ensure that any actions you perform are appropriately reversed in the removing event to prevent memory leaks occuring.

Docking Drag & Drop Events

Drag & Drop Events: DoDragDropEnd
DoDragDropQuit

Event Firing
At the end of a page drag and drop action within the docking system one of these events will be fired. If the drop occurred with success then the *DoDragDropEnd* event will be fired to indicate success. If the drag operation was cancelled for any reason then the *DoDragDropQuit* event will be fired instead. Note that only one of these events is fired at the end of an operation and never both.

External Event Usage
The most likely scenario for needing these events is if you are starting off a drag and drop operation manually from outside the docking system and then need to know when the operation has been completed. For example, you might have a *TreeView*that contains a set of nodes. When the user starts dragging one of these tree nodes you want to create a floating window as part of the docking system and allow that floating window to be dragged just like any other docking content. The creation of a new docking page and associated floating window is very simple and shown in example code below. Once the floating window is created you would call the docking manager *DoDragDrop* method in order to request that a drag operation occur with the newly created floating window. You will need to know when this drag operation completes so you can update the status of your *TreeView*control. By hooking both the *DoDragDropEnd* and *DoDragDropQuit* you can update as needed when the operation completes. The following sample code shows how this might look in practice.

```
protected override void OnMouseMove(MouseEventArgs e) {
  Point pt = new Point(e.X, e.Y);

  // Has a node been selected from dragging?
  if (_dragNode != null)
  {

    // Create a new page for the docking system
    KryptonPage dp =  new KryptonPage("New Page");

    // Create a floating window that contains the new page
    KryptonDockingFloatingWindow fw = kryptonDockingManager.AddFloatingWindow("Floating", new KryptonPage[] { dp });

    // Spin message loop so new window appears
    Application.DoEvents();

    // We want to know when the drag drop operating is finished
    kryptonDockingManager.DoDragDropEnd += new EventHandler(kryptonDockingManager_DoDragDropFinished);
    kryptonDockingManager.DoDragDropQuit += new EventHandler(kryptonDockingManager_DoDragDropFinished);

    // Drag the new floating window around
    kryptonDockingManager.DoDragDrop(MousePosition, pt, dp, fw);
  }

  base.OnMouseMove(e);

}



private void kryptonDockingManager_DoDragDropFinished(object sender, EventArgs e)
{

  // Remember to unhook from no longer needed events
  kryptonDockingManager.DoDragDropEnd -= new EventHandler(kryptonDockingManager_DoDragDropFinished);
  kryptonDockingManager.DoDragDropQuit -= new EventHandler(kryptonDockingManager_DoDragDropFinished);

  // Drag has finished so set drag node back to null
  _dragNode = null;
}
```

Ribbon

Learn more about the capabilities of the *KryptonRibbon* using the following sections.

Overview

Overview

Operation

Tabs
Groups
Group Containers
Group Items
Contextual Tabs
Application Button
Quick Access Toolbar
KeyTips & Keyboard Access
ButtonSpecs

Events

Control Events
Item Events

Standalone Controls

KryptonGallery

Ribbon Overview

The *Ribbon* control is used to emulate the user interface found in many of the the *Microsoft Office 2007* applications such as *Word* and *Excel*. Included are all the main features you would expect such as chrome integration, contextual tabs, full keyboard access and much more.

Ribbon Values Properties

Figure 1 shows a list of the *Values* category properties related to the ribbon control. Following is a description of how to use each of the properties along with relevant links to more detailed explanations.

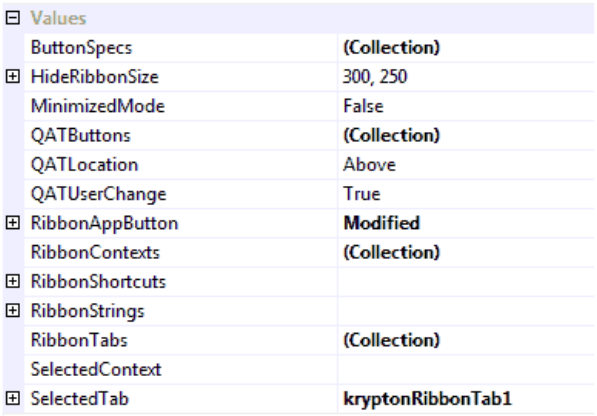


Figure 1 - Ribbon Values Properties

ButtonSpecs

You can use the *ButtonSpecs* collection to add extra buttons onto the tab bar of the ribbon. The tab bar is the area where tab headers and contextual tab headers appear. This is a common requirement as many applications will want to place a help button on the right side of the tab bar just like that present in *Word 2007* and *Excel 2007*. You can get more details about using this property at the following link.

HideRibbonSize

The ribbon control takes up quite a fair chunk of screen real estate and once the user resizes the owning window down to a particular size it makes more sense to hide the ribbon entirely. This property is used to determine at what height and width the ribbon will automatically hide itself. Once the owning window is resized bigger than this size then it will become visible again.

MinimizedMode

By default the ribbon is fully visible but you can toggle the control into a minimized mode in order to save space. The user can do this at runtime by double clicking one of the tab headers or using the *CTRL + F1* key combination. Use this property to toggle the mode using code or at design time.

QATButtons QATLocation
QATUserChange

These three properties control the quick access toolbar that by default is placed at the top of the ribbon control adjacent to the application button. You would typically use the quick access toolbar for common actions that the user always needs access to whatever tab is currently selected. For more information about these properties using this link.

RibbonAppButton

The application button is the large round button that appears at the top left of the ribbon control. This button is always displayed and when possible integrated into the caption area of the owning *Form* instance. When using the ribbon you should not use a menu or toolbar strip and so this button acts as the equivalent of the *File* menu that would usually appear in a typical application. You can get more details about the application button and these properties at the following link.

RibbonContexts

If using contextual tabs in your application then you need to use this collection property to define each of the different contexts required. Once you have defined the contexts you should update the *ContextName* property of each relevant tab to associate it with the appropriate context definition. Finally update the *SelectedContext* property, described below, to define the currently displayed context. For more information use the following link.

RibbonShortcuts

This is a compound property that has a set of child properties for defining individual shortcuts. Update the child values in order to define which key combinations are used to enter keyboard mode and to switch to/from minimized modes.

RibbonStrings

All display strings used by the control are placed inside this compound property. This is important as it allows the developer to localize the text strings for different languages by changing the form level culture. Most of the strings inside this compound property related to the context menus that the ribbon shows.

RibbonTabs

This is the root property used to define individual tabs for the ribbon. As a collection property you can use it to add/remove and update

tab instances. Each tab itself has a *Groups* property for defining the contents of the tab when it is selected. For more detail use the following link.

SelectedContext

When using contextual tabs you use this property to define which contexts are currently displayed. The value is a comma separated list of context names where the contexts are displayed in the same order as they appear in the property. For mode information use the following link.

SelectedTab

Use this property to define which of the displayed tabs should be the currently selected tab. If you attempt to set a tab that is not possible to be displayed then it will throw an exception.

Ribbon Visuals Properties

Figure 2 shows a list of the *Visuals* category properties related to the ribbon control. Following is a description of how to use each of the properties along with relevant links to more detailed explanations.

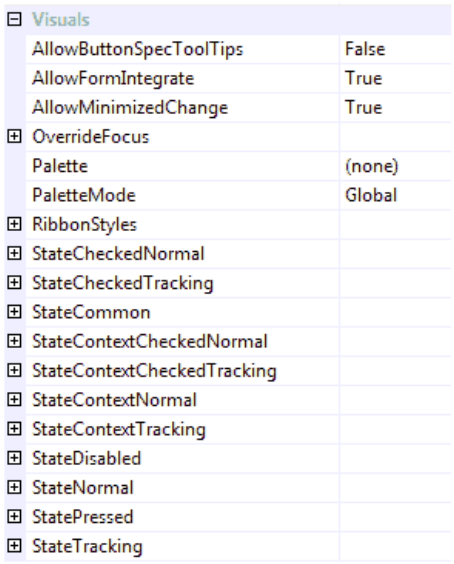


Figure 2 - Ribbon Visuals Properties

AllowButtonSpecTooltips

By default the Ribbon will not show any tool tips when you hover the mouse over the user defined button specifications that appear on the tabs line of the control. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

AllowFormIntegrate

Under *Vista* with the *Aero* theme enabled there are two choices for how the ribbon integrates into the custom chrome of a *KryptonForm*. If you leave the default value of *True* in place then it will show with glass chrome and have the same appearance as *Office 2007* applications under *Vista*. Change this property to *False* and you revert to the same appearance as under XP. Figure 3 shows the two settings, *True* on the left and *False* on the right.

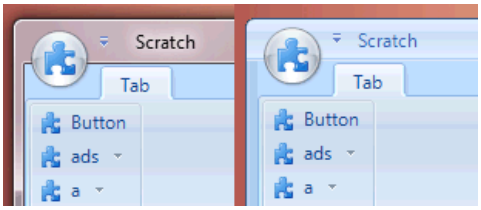


Figure 3 - Ribbon Form Integration

AllowMinimizedChange

Determines if the user is allowed to change the minimized mode of the ribbon at runtime. They can do this using the keyboard combination *Ctrl+F1*, double clicking a page header or by using one of the appropriate context menu options. Set this property to False to prevent the user changing the mode. Note that this property does not prevent programmatic changing of the *MinimizedMode* property.

RibbonStyles

This compound property contains a list of the different styles that can be changed. For example you can change the background style from the default of *Panel - Client* if you prefer a different background that is appropriate for your application. Alternatively you could alter the display style of the group buttons, cluster buttons or various other ribbon elements.

Ten States

As with all the *Krypton* controls, each possible state of the control has a set of properties that can be used to customize the appearance. For the ribbon there are ten possible states that various elements can be in but not all the elements use all the states. The ten states are named *StateDisabled*, *StateNormal*, *StateTracking*, *StatePressed, StateCheckedNormal, StateCheckedTracking, StateContextCheckedNormal, StateContextCheckedTracking, StateContextNormal and StateContextTracking*.

Common State

To speed up the customization process an extra *StateCommon* property has been provided. The settings from this are used if no override has been defined for the state specific entry. Note that the specific state values always take precedence and so if you define a property in both the *StateNormal* and *StateCommon* sets then the *StateNormal* value will be used whenever the control is in the *Normal* state. Only if the *StateNormal* value is not overridden will it look in *StateCommon*.

Override State

There is an additional state related property called *OverrideFocus* is used to alter the appearance of the control when it has the focus. Notice that the property starts with the *Override* prefix instead of the usual *State*. This is because it does not relate to a specific control state such as *Normal* or *Tracking*. Instead it is applied to any of the other states and is used to override the appearance that would otherwise be shown.

Ribbon Tabs

The ribbon is similar to a *TabControl* in that it has a set of tabs that are represented via headers. Headers are shown at the top of the ribbon control and presented from left to right. The ordering of the headers matches the ordering of non-contextual tabs in the *RibbonTabs* collection. Note that contextual tabs are always displayed at the end of the headers in related groupings and so contextual tabs do not honor the ordering of the instances inside the *RibbonTabs* collection.
To view the properties for a tab at design time you just need to left click the tab header in the ribbon control and then use the *Properties* window to view and modify them. Alternatively you can use the *Document Outline* window to select the tab instance of interest and then use the *Properties* window.

Ribbon Tab Properties
You can see in Figure 1 the ribbon properties relating to an individual tab.

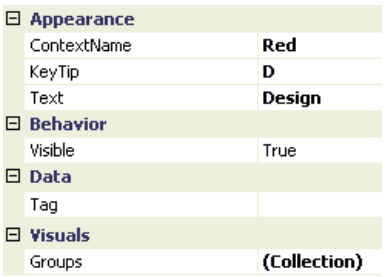


Figure 1 - Ribbon Tab Properties

ContextName
A non-contextual tab will leave this property empty because the tab does not belong to a context. If you want to associate this tab with a *RibbonContext* then you should set the same name in this property as is defined in the *RibbonContext.ContextName*. In that case the tab will only be displayed is the *SelectedContext* property of the ribbon control contains the *RibbonContext.ContextName*.

KeyTip
When entering *KeyTips* mode each visible tab will show a *KeyTip* pop up with the string defined in this property. You should provide a unique *KeyTip* value for each tab in the ribbon control to ensure that all the tabs can be accessed via the keyboard. If multiple tabs have the same *KeyTip* then pressing the *KeyTip* will action the first tab that matches.

Text
This is the display text for the tab and will appear in the header of the ribbon control when the tab is visible. The property is localizable so that you can define different values for different culture settings. A value must always be provided for this property, if you attempt to set an empty string it will default to *Tab* instead.

Visible
A default value of *True* ensures the tab is make visible. Note that a contextual tab will not be visible if the associated context is not selected regardless of the *Visible* property value.

Groups
Each tab instance contains a collection of groups that are displayed below the tab header when the tab is selected. Use this collection to add, remove and modify the displayed groups for the tab. To manipulate the set of child groups at design time you are recommend to use the helper elements that appear on the ribbon control and not to use the collection editor.

Tag
Associate application specific information with the object instance by using this property.

Ribbon Groups

A ribbon tab can have any number of child ribbon group instances. They are positioned from left to right within the tab area and automatically sized according to the content they contain. Each group can be in either a normal or collapsed state. In normal state the group shows its content and has a title area at the bottom of the group area. When collapsed the group does not show any content and becomes a button that when pressed shows the content in a pop up window. The ribbon control will automatically determine which of the two states the group should be in when it auto calculates the sizing of groups based on the available space.
To view the properties for a group at design time you just need to left click the group title area in the ribbon control and then use the *Properties* window to view and modify them. Alternatively you can use the *Document Outline* window to select the group instance of interest and then use the *Properties* window.

Ribbon Group Properties
You can see in Figure 1 the ribbon properties relating to an individual group.

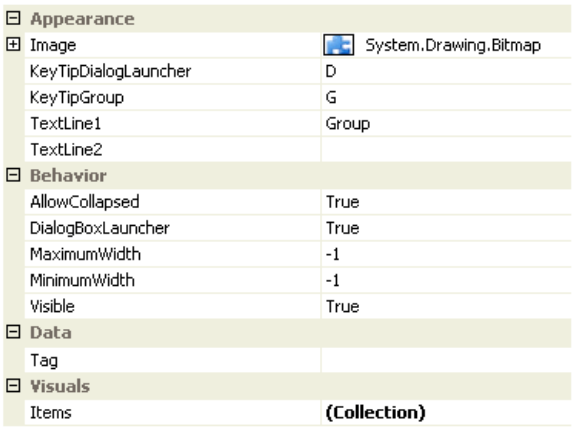


Figure 1 - Ribbon Group Properties

Image
An image is only displayed when the group is in the collapsed state and presented as a button that can be pressed in show the content in a pop up window. You should always provide an image that is 16 x 16 pixel as the image will always be resized to draw at that size. It is not possible to provide a *null* image as a default image of a blue jigsaw piece will be shown if you do not provide an image yourself.

KeyTipDialogLauncher
This property is only applicable when the group is in the normal mode and the *DialogBoxLauncher* property is defined as *True*. In these circumstances you need to define the *KeyTip* that is displayed so the user can use the keyboard to select the dialog box launcher button. This is the button that appears in the bottom right of the group title area.

KeyTipGroup
In *KeyTips* mode when the group is collapsed a *KeyTip* is needed to cause the collapsed group to be selected and a pop up window shown with the contents of the group. If the group is not collapsed then this property is ignored because all parts of the group are already visible and accessible.

TextLine1
TextLine2
In normal mode *TextLine1* and *TextLine2* are concatenated together with a space between them as the title text of the group. In collapsed mode the *TextLine1* and *TextLine2* are shown as two separate lines of text underneath the Image.

AllowCollapsed
Use this property to inform the ribbon control if the group is allowed to become collapsed during the auto sizing of the groups. By default this property is *True* and so allows groups to become collapsed but you can define it as *False* if you application requires the group contents to always be shown.

DialogBoxLauncher
The dialog box launcher button is displayed at the bottom right of the group title area. It is intended to be used so the user can press it and have a modal dialog appear presenting extra options for the user. If you prefer to remove this button from the group then set this property to *False*.

MaximumWidth
MinimumWidth
These properties default to -1 and so are ignored in effect. When defined with positive values these are applied to the sizing of the group and can be used to enforce limits.

Visible
All groups are by default set to visible but you can hide them from display by setting this to *False*.

Items
Each group instance contains a collection of group container items that are displayed from left to right inside the group. Use this collection to add, remove and modify the group containers for the group. To manipulate the set of child containers at design time you are recommend to use the helper elements that appear on the ribbon control and not to use the collection editor.

Tag
Associate application specific information with the object instance by using this property.

Ribbon Group Containers

A ribbon group can only contain child items that are group containers. Individual group containers have their own restrictions on what types of child item they can themselves contain. Different group containers have different ways of laying out the child items they contain and different rules on how to automatically resize when limited space is available. By choosing the correct type of group container the developer can therefore specify exactly how auto sizing will effect the items it contains.

When editing a ribbon group at design time the collection editor will present you with a list of the available group containers that can be added. The preferred method of designing the ribbon is to use the helper elements that are show in the client area of the ribbon itself. If you click the helper element at the right hand side of the group client area it will show a context menu with the different types of group container allowed. On selecting one of these it will add a new instance with some default items inside the container.

Following are a detailed description of the different types of group container.

Group Container Separator

The group separator is the simplest of all the group containers and should be used to draw a visual separator between other group containers. This container does not allow you to add any child items. Figure 1 shows an example of a group with six group separators at design time. On the right hand side you will notice the *New*block, this is used at design time to add another group container and when pressed will show a context menu with the types of container available for adding. At runtime this helper element is not present. There are only two properties of interest exposed by the group separator as can be seen in Figure 2.

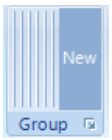


Figure 1 - Group Separator Design Time Appearance

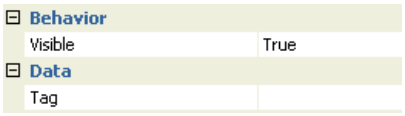


Figure 2 - Group Separator Properties

Visible

Use this property to specify if the group triple should be visible at runtime.

Tag

Associate application specific information with the object instance by using this property.

Group Container Triple

As the name suggests the group triple container is used to display a maximum of three child items. Figure 3 shows an example of a group at design time that contains a single group triple instance with three child group button items. The group triple at runtime has no user interface but at design time shows the dark blue area around the three buttons. This dark blue area can be used to select the group triple container instance and allow you to update the properties of the instance in the *Properties Window*. On the right hand side you will notice the *New*block, this is used at design time to add another group container and when pressed will show a context menu with the types of container available for adding. At runtime neither the dark blue or *New*button will be present. Figure 4 shows a list of the properties you can modify for the group triple instance.

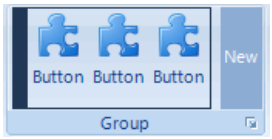


Figure 3 - Group Triple Design Time Appearance

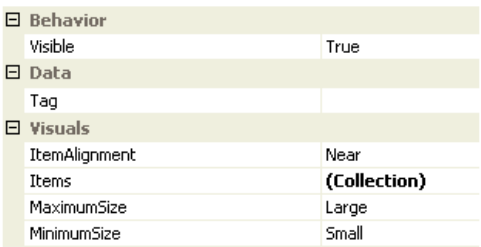


Figure 4 - Group Triple Properties

Visible
Use this property to specify if the group triple should be visible at runtime.

Tag
Associate application specific information with the object instance by using this property.

ItemAlignment
Determines how items are horizontally aligned when in the medium and small settings.

Items
A collection property that contains a maximum of three child items.

MaximumSize MinimumSize
These two properties are used to determine the maximum and minimum sizing used for the group at runtime. The maximum size is always the size used if there is enough room for the group to size fully. If there is not enough room then each intermediate size is used until the minimum is reached. The group triple never sizes its child items smaller than the minimum setting. Figure 5 shows the group triple at runtime in each of the three possible size values of *Large*, *Medium* and *Small*.

Figure 5 - Group Triple Sizing - Large, Medium and Small

Group Container Lines

The group lines container is used to layout child items over either two or three horizontal lines. Figure 6 shows an example of a group at design time that contains a single group lines instance with four child group button items. The group lines at runtime has no user interface but at design time shows the dark blue area around the three buttons. This dark blue area can be used to select the group lines container instance and allow you to update the properties of the instance in the *Properties Window*. On the right hand side you will notice the *New*block, this is used at design time to add another group container and when pressed will show a context menu with the types of container available for adding. At runtime neither the dark blue or *New*button will be present. Figure 7 shows a list of the properties you can modify for the group line instance.

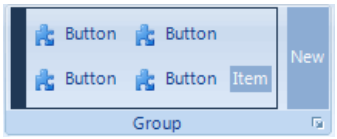


Figure 6 - Group Lines Design Time Appearance

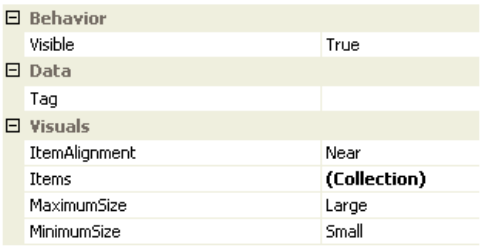


Figure 7 - Group Lines Properties

Visible
Use this property to specify if the group lines should be visible at runtime.

Items
A collection property that contains a zero or more child items.

MaximumSize MinimumSize
These two properties are used to determine the maximum and minimum sizing used for the group at runtime. The maximum size is always the size used if there is enough room for the group to size fully. If there is not enough room then each intermediate size is used until the minimum is reached. The group triple never sizes its child items smaller than the minimum setting. Figure 8 shows the group triple at runtime in each of the three possible size values of *Large*, *Medium* and *Small*.

Figure 8 - Group Line Sizing - Large, Medium and Small

Tag
Associate application specific information with the object instance by using this property.

Ribbon Group Items

There are several types of group item that can be placed in the various group containers. Not all item types can be placed in all the container types so read the documentation on the individual item types for a list of the compatible containers.

Adding new item instances at design time is very easy. Just use the element helpers called *New*that are placed inside the group containers. If only a single type of item is allowed that item type will be created and added immediately. If multiple item types are allowed then a context menu will appear presenting all the different options you can choose. To edit an existing item just left click it with the mouse and the *Properties Window*will show the details are editing.
Following are the different types of group items.

- *GroupItemLabel*
- *GroupItemButton*
- *GroupItemCheckBox*
- *GroupItemCluster*
- *GroupItemClusterButton*
- *GroupItemClusterColorButton*
- *GroupItemColorButton*
- *GroupItemComboBox*
- *GroupItemCustomControl*
- *GroupItemDateTimePicker*
- *GroupItemGallery*
- *GroupItemNumericUpDown*
- *GroupItemRadioButton*
- *GroupItemRichTextBox*
- *GroupItemTextBox*
- *GroupItemTrackBar*

Group Item Label

The label item can be added to a group triple or a group lines container. Use this item when you need to present static information or to label another item that is adjacent. Figure 1 shows the list of all properties exposed by the item label instance.

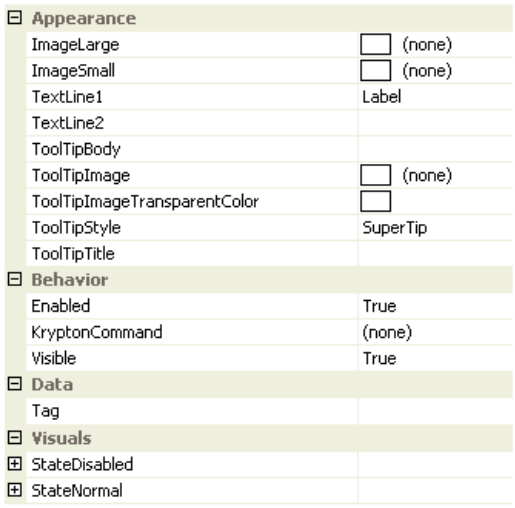


Figure 1 - Group Item Label Properties

ImageLarge
ImageSmall
By default the label item does not show any images. If you need an image shown when the label is the full height of the group content area then use the *ImageLarge* property. All other cases use the *ImageSmall* if it has been specified.

TextLine1 TextLine2
When the label is inside a container that displays it the full height of the group content area the *TextLine1* and *TextLine2* strings are shown on two separate lines underneath the *ImageLarge*. In all other cases the *TextLine1* and *TextLine2* are concatenated together with a space between them for showing horizontally after the *ImageSmall*.

ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle
When the user hovers the mouse over the label instance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.
Enabled
Used to define if the label is enabled or disabled at runtime.

KryptonCommand
Attached command that is used as a source of state.

Visible
Use this property to specify if the label should be visible at runtime.

StateDisabled
StateNormal
You can modify the colors used for drawing the label text by using these properties. All other types of item and container cannot be customized on a per instance basis. However, the label has the extra customization ability because it allows state information to be related to the user more effectively. For example, you might set text color to green when your application is working correctly and then change the text to red when a problem occurs.

Tag
Associate application specific information with the object instance by using this property.

Group Item Button

You can add a group button to either the group triple or the group lines container. Use the *ButtonType* property to define the type of button operation you require. Figure 1 shows the list of all properties exposed by the group button item.

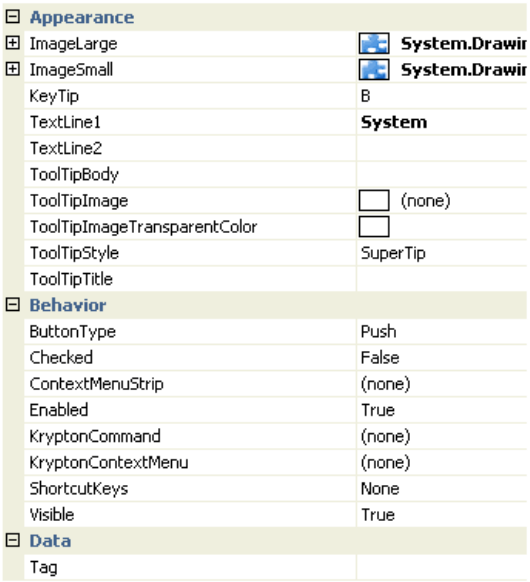


Figure 1 - Group Item Button Properties

ImageLarge
ImageSmall

The button item must always have valid images defined for the ImageLarge and ImageSmall. If you assign *null* to the properties then it will automatically revert to a default blue jigsaw piece image instead. When the button is the full height of the group content area the *ImageLarge* property is used, all other cases use the *ImageSmall* property.

KeyTip

When *KeyTips* are displayed this property defines the KeyTip for the button instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

TextLine1 TextLine2

When the button is inside a container that displays it the full height of the group content area the *TextLine1* and *TextLine2* strings are shown on two separate lines underneath the *ImageLarge*. In all other cases the *TextLine1* and *TextLine2* are concatenated together with a space between them for showing horizontally after the *ImageSmall*.

ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle

When the user hovers the mouse over the button instance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.

ButtonType

Determines the appearance and operation of the button. Possible values include:-

- Push - A traditional push button
- Check - Toggles between checked and unchecked
- DropDown - Displays a context menu strip when pressed
- Split - Split between a traditional push button and a drop down area

Checked

This property is only used when the *ButtonType* is defined as *Checked*. It determines if the button should be drawn with the checked appearance. The value will automatically be toggled between *True* and *False* when the user clicks the button and *ButtonType* is *Checked*.

ContextMenuStrip

When the user clicks a button of type *DropDown* or the split portion of a *Split* button this property is used if defined and the *KryptonContextMenu* property is null. This property is provided as a parameter to the buttons *DropDown* event so that the developer can customize the context menu strip before it is displayed.

Enabled

Used to define if the button is enabled or disabled at runtime.

KryptonCommand
Attached command that is used as a source of state.

KryptonContextMenu
When the user clicks a button of type *DropDown* or the split portion of a *Split* button this property is used if defined, otherwise the *ContextMenuStrip* is used. This property is provided as a parameter to the buttons *DropDown* event so that the developer can customize the context menu before it is displayed.

ShortcutKeys
Define this property if you would like your application to have a shortcut key combination that invokes the button. For example, buttons used for cut/copy/paste would be defined with shortcut keys so the user can invoke those actions without moving focus away from the current control.

Visible
Use this property to specify if the button should be visible at runtime.

Tag
Associate application specific information with the object instance by using this property.

Events

DropDown
A cancelable event than allows you to customize the drop down context menu before it is displayed.

Click
Occurs when the button has been pressed.

Group Item CheckBox

You can add a check box to either the group triple or the group lines container. Use the *CheckState* property to define the checked state of the check box. Figure 1 shows the list of all properties exposed by the group check box item.

Appearance	
KeyTip	C
TextLine1	CheckBox
TextLine2	
ToolTipBody	
ToolTipImage	(none)
ToolTipImageTransparentColor	
ToolTipStyle	SuperTip
ToolTipTitle	
Behavior	
AutoCheck	True
Checked	False
CheckState	Unchecked
Enabled	True
KryptonCommand	(none)
ShortcutKeys	None
ThreeState	False
Visible	True
Data	
Tag	

Figure 1 - Group Item CheckBox Properties

KeyTip
When *KeyTips* are displayed this property defines the *KeyTip* for the check button instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

TextLine1 TextLine2
When the check box is inside a container that displays it the full height of the group content area the *TextLine1* and *TextLine2* strings are shown on two separate lines underneath the check box image. In all other cases the *TextLine1* and *TextLine2* are concatenated together with a space between them for showing horizontally after the check box image.

ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle
When the user hovers the mouse over the check box instance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.

AutoCheck
When defined as *True* this property will allow the user to changed the check state by using the mouse to click the control instance.

Checked
A boolean property this indicates if the check box is currently checked or not. If you require a three state use of the check box then you should use the *CheckState* property as that allows an intermediate state to be defined. For regular use as either a checked or not checked state control then you can use this *Checked* property.

CheckState
Use this property to define one of the three possible display states of the check box. The available states are *Checked*, *Unchecked* or *Indeterminate*. The checked state will result in a tick appearing inside the check box, the unchecked state shows just an empty check box and the indeterminate state a square inside the check box.

Enabled
Used to define if the check box is enabled or disabled at runtime.

KryptonCommand
Attached command that is used as a source of state.

ShortcutKeys
Define this property if you would like your application to have a shortcut key combination that invokes the check box.

ThreeState
Should selecting the control rotate around the three possible states or should it rotate around just the checked/unchecked state.

Visible

Use this property to specify if the check box should be visible at runtime.

Tag
Associate application specific information with the object instance by using this property.

Events

Click
Occurs when the check box has been pressed.

CheckedChanged
Occurs when the *Checked* property changes value.

CheckStateChanged
Occurs when the *CheckState* property changes value.

Group Item Cluster

A cluster item can be added as a child of a group lines container but cannot be added to any other type of container. It has no user interface of its own but instead acts as a container for cluster button items. Note that this cluster item cannot be added as a child of a group. Figure 1 shows the properties for the cluster item.

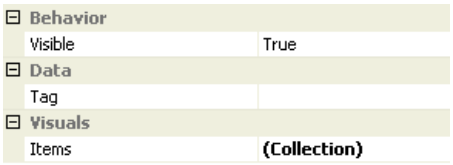


Figure 1 - Group Item Cluster Properties

Visible

Use this property to specify if the cluster should be visible at runtime.

Items

This is a collection property that allows child cluster button items to be added/removed and modified.

Tag

Associate application specific information with the object instance by using this property.

Group Item Cluster Button

You can only add a cluster button as a child of a cluster instance, they cannot be placed inside the group triple or group lines containers. Figure 1 shows the properties of the cluster button which are almost the same as those of the more general group button item.

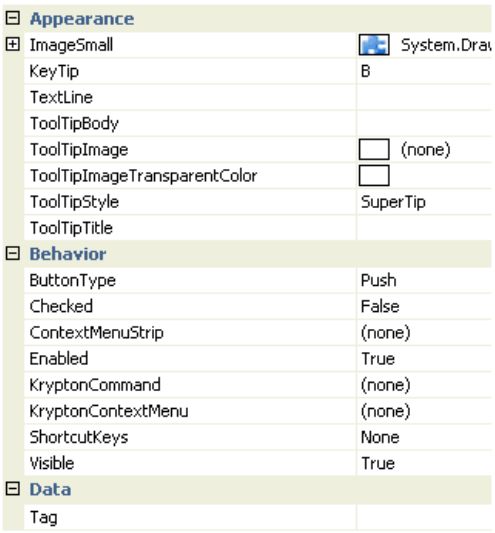


Figure 1 - Group Item Cluster Button Properties

ImageSmall
The cluster button item must always have a valid image defined. If you assign *null* to the *Image* property then it will automatically revert to a default blue jigsaw piece image instead.

KeyTip
When *KeyTips* are displayed this property defines the *KeyTip* for the cluster button instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

TextLine
Assign the display text for the cluster button to the *TextLine* property.

ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle
When the user hovers the mouse over the button instance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.

ButtonType
Determines the appearance and operation of the button. Possible values include:-

- Push - A traditional push button
- Check - Toggles between checked and unchecked
- DropDown - Displays a context menu strip when pressed
- Split - Split between a traditional push button and a drop down area

Checked
This property is only used when the *ButtonType* is defined as *Checked*. It determines if the cluster button should be drawn with the checked appearance. The value will automatically be toggled between *True* and *False* when the user clicks the cluster button and *ButtonType* is *Checked*.

ContextMenuStrip
When the user clicks a button of type *DropDown* or the split portion of a *Split* button this property is used if defined and the *KryptonContextMenu* property is null. This property is provided as a parameter to the buttons *DropDown* event so that the developer can customize the context menu strip before it is displayed.

Enabled
Used to define if the button is enabled or disabled at runtime.

KryptonCommand
Attached command that is used as a source of state.

KryptonContextMenu
When the user clicks a button of type *DropDown* or the split portion of a *Split* button this property is used if defined, otherwise the *ContextMenuStrip* is used. This property is provided as a parameter to the buttons *DropDown* event so that the developer can customize the context menu before it is displayed.

ShortcutKeys
Define this property if you would like your application to have a shortcut key combination that invokes the cluster button. For example, cluster buttons used for cut/copy/paste would be defined with shortcut keys so the user can invoke those actions without moving focus away from the current control.

Visible
Use this property to specify if the cluster button should be visible at runtime.

Tag
Associate application specific information with the object instance by using this property.

Events

DropDown
A cancelable event than allows you to customize the drop down context menu before it is displayed.

Click
Occurs when the button has been pressed.

Group Item Cluster Color Button

You can only add a cluster color button as a child of a cluster instance, they cannot be placed inside the group triple or group lines containers. Figure 1 shows the properties of the cluster color button which are almost the same as those of the more general group cluster button item.

Appearance	
EmptyBorderColor	DarkGray
ImageSmall	System.Drawing.Bitmap
KeyTip	B
RecentColors	**Color[] Array**
SelectedColor	Red
SelectedRect	0, 12, 16, 4
TextLine	
ToolTipBody	
ToolTipImage	(none)
ToolTipImageTransparentColor	
ToolTipStyle	SuperTip
ToolTipTitle	
Behavior	
AutoRecentColors	True
ButtonType	Split
Checked	False
Enabled	True
KryptonCommand	(none)
MaxRecentColors	10
SchemeStandard	OfficeStandard
SchemeThemes	OfficeThemes
ShortcutKeys	None
Visible	True
VisibleMoreColors	True
VisibleNoColor	True
VisibleRecent	True
VisibleStandard	True
VisibleThemes	True
Data	
Tag	

Figure 1 - Group Cluster Color Button Properties

EmptyBorderColor
Color to draw the border of the selected rect when the selected color is *Color.Empty*.

ImageSmall
The cluster color button item must always have a valid image defined. If you assign *null* to the *Image* property then it will automatically revert to a default font image instead.

KeyTip
When *KeyTips* are displayed this property defines the KeyTip for the button instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

RecentColors
Array of colors to show in the recent colors section of the context menu.

SelectedColor
The currently selected color which can be the *Color.Empty* value.

SelectedRect
The rectangle of the displayed image that should be drawn in the selected color. If the selected color is *Color.Empty* then the border of this rectangle is drawn in the *EmptyBorderColor* instead.

TextLine
Assign the display text for the cluster color button to the *TextLine* property.

ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle
When the user hovers the mouse over the button instance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.

AutoRecentColors

Should the *RecentColors* array be automatically updated to include newly selected colors that are not contained in either the standard or theme sections.

ButtonType
Determines the appearance and operation of the button. Possible values include:-

- Push - A traditional push button
- Check - Toggles between checked and unchecked
- DropDown - Displays a context menu strip when pressed
- Split - Split between a traditional push button and a drop down area

Checked
This property is only used when the *ButtonType* is defined as *Checked*. It determines if the button should be drawn with the checked appearance. The value will automatically be toggled between *True* and *False* when the user clicks the button and *ButtonType* is *Checked*.

Enabled
Used to define if the button is enabled or disabled at runtime.

KryptonCommand
Attached command that is used as a source of state.
MaxRecentColors
Limit placed on the size of the *RecentColors* array.
SchemeStandard
SchemeThemes
Enumeration values indicating the predefined sets of colors to show in the standard and themed sections of the context menu.

ShortcutKeys
Define this property if you would like your application to have a shortcut key combination that invokes the button. For example, buttons used for cut/copy/paste would be defined with shortcut keys so the user can invoke those actions without moving focus away from the current control.

Visible
Use this property to specify if the button should be visible at runtime.

VisibleMoreColors VisibleNoColor
VisibleRecent VisibleStandard
VisibleThemes
Should the relevant section of the context menu be displayed.

Tag
Associate application specific information with the object instance by using this property.

Events

DropDown
A cancelable event than allows you to customize the drop down context menu before it is displayed.

Click
Occurs when the button has been pressed.

MoreColors
Occurs when the user selects the *MoreColors* menu option.

SelectedColorChanged
Occurs when the value of the *SelectedColor* property changes.

TrackColor
As the user tracks over different colors this event fires so you can provide instance feedback on the effect this would have if selected.

Group Item Color Button

You can add a group color button to either the group triple or the group lines container. Use the *ButtonType* property to define the type of button operation you require. Figure 1 shows the list of all properties exposed by the group color button item.

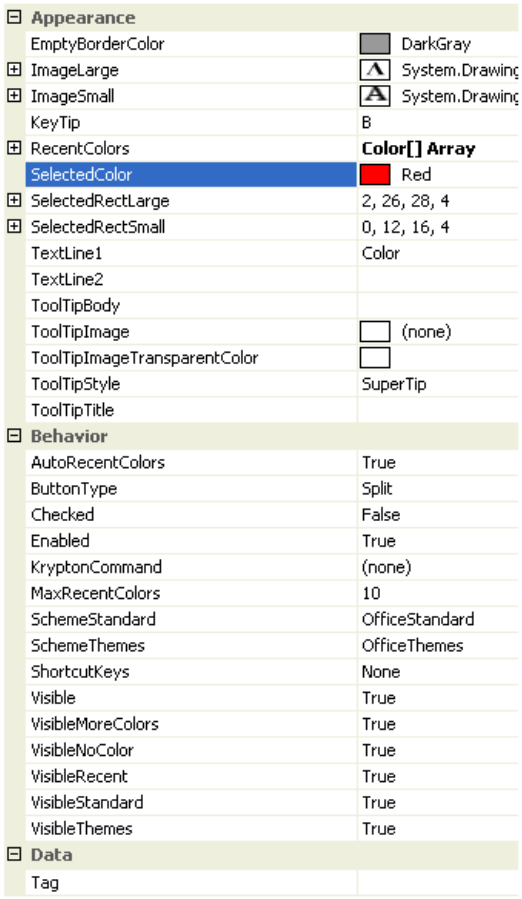


Figure 1 - Group Item Color Button Properties

EmptyBorderColor
Color to draw the border of the selected rect when the selected color is *Color.Empty*.

ImageLarge ImageSmall
If you need an image shown when the label is the full height of the group content area then use the *ImageLarge* property. All other cases use the *ImageSmall* if it has been specified.

KeyTip
When *KeyTips* are displayed this property defines the KeyTip for the button instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

RecentColors
Array of colors to show in the recent colors section of the context menu.

SelectedColor
The currently selected color which can be the *Color.Empty* value.

SelectedRect
The rectangle of the displayed image that should be drawn in the selected color. If the selected color is *Color.Empty* then the border of this rectangle is drawn in the *EmptyBorderColor* instead.

TextLine1 TextLine2
When the color button is inside a container that displays it the full height of the group content area the *TextLine1* and *TextLine2* strings are shown on two separate lines underneath the color button image. In all other cases the *TextLine1* and *TextLine2* are concatenated together with a space between them for showing horizontally after the color button image.

ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle
When the user hovers the mouse over the button instance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many

bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.

AutoRecentColors
Should the *RecentColors* array be automatically updated to include newly selected colors that are not contained in either the standard or theme sections.

ButtonType
Determines the appearance and operation of the button. Possible values include:-

- Push - A traditional push button
- Check - Toggles between checked and unchecked
- DropDown - Displays a context menu strip when pressed
- Split - Split between a traditional push button and a drop down area

Checked
This property is only used when the *ButtonType* is defined as *Checked*. It determines if the button should be drawn with the checked appearance. The value will automatically be toggled between *True* and *False* when the user clicks the button and *ButtonType* is *Checked*.

Enabled
Used to define if the button is enabled or disabled at runtime.

KryptonCommand
Attached command that is used as a source of state.
MaxRecentColors
Limit placed on the size of the *RecentColors* array.
SchemeStandard
SchemeThemes
Enumeration values indicating the predefined sets of colors to show in the standard and themed sections of the context menu.

ShortcutKeys
Define this property if you would like your application to have a shortcut key combination that invokes the button. For example, buttons used for cut/copy/paste would be defined with shortcut keys so the user can invoke those actions without moving focus away from the current control.

Visible
Use this property to specify if the button should be visible at runtime.

VisibleMoreColors VisibleNoColor
VisibleRecent VisibleStandard
VisibleThemes
Should the relevant section of the context menu be displayed.

Tag
Associate application specific information with the object instance by using this property.

Events

DropDown
A cancelable event than allows you to customize the drop down context menu before it is displayed.

Click
Occurs when the button has been pressed.

MoreColors
Occurs when the user selects the *MoreColors* menu option.

SelectedColorChanged
Occurs when the value of the *SelectedColor* property changes.

TrackColor
As the user tracks over different colorsthis event fires so you can provide instance feedback on the effect this would have if selected.

Group Item ComboBox

Use this element to add combo box control to the ribbon. The set of available properties is shown in Figure 1.

Appearance	
DropDownStyle	DropDown
KeyTip	X
Text	
Behavior	
ContextMenuStrip	(none)
DropDownHeight	106
DropDownWidth	143
Enabled	True
ItemHeight	**13**
MaxDropDownItems	8
MaxLength	0
ShortcutKeys	None
Sorted	False
Visible	True
Data	
DataSource	(none)
DisplayMember	(none)
Items	**(Collection)**
Tag	
ValueMember	
Layout	
MaximumSize	121, 0
MinimumSize	121, 0
Misc	
AutoCompleteCustomSourc	**(Collection)**
AutoCompleteMode	None
AutoCompleteSource	None
FormatString	
FormattingEnabled	**False**
Visuals	
AllowButtonSpecToolTips	False
ButtonSpecs	**(Collection)**

Figure 1 - Group Item ComboBox Properties

KeyTip

When *KeyTips* are displayed this property defines the *KeyTip* for the custom control instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

Enabled

Used to define if the combo box is enabled or disabled at runtime.

ShortcutKeys

Define this property if you would like your application to have a shortcut key combination that sets focus to the combo box.

Visible

Use this property to specify if the combo box should be visible at runtime.

MaximumSize MinimumSize

These two properties are very important and allow you to control the width of the combo box. By default they have the same value giving a fixed width to the control, but you can alter the values so that they specify a valid range of widths. When the control is positioned it is asked for its preferred size which is then constrained by these two properties.

ButtonSpecs

You can add extra buttons to the combo box by modifying the *ButtonSpecs* collection. Each *ButtonSpec* entry in the collection describes a single button for display. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances.

AllowButtonSpecTooltips

By default the combo box will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

Tag

Associate application specific information with the object instance by using this property.

Other Properties

All the other properties displayed in Figure 5 are just properties from the underlying *KryptonComboBox* instance and exposed via this element for ease of access. For more information about these properties see the standard windows documentation for the *ComboBox* control.

Hidden Property

In order to gain access to the underlying *KryptonComboBox* instance you can use the hidden property called *ComboBox*. This is not displayed at design time and not persisted but can be accessed via code at runtime. This is useful if you need to access a property that is not exposed via this ribbon element.

Group Item CustomControl

You can add any control to be displayed in the ribbon by creating a placeholder for the control with this custom control item. Then inside the load event for the owning *Form* you can then associate the control instance with the place holder by assigning it to the *CustomControl* property. See the *Ribbon Custom Controls* sample for examples of this and the associated source code. Figure 1 shows the list of all properties exposed by the group custom control item.

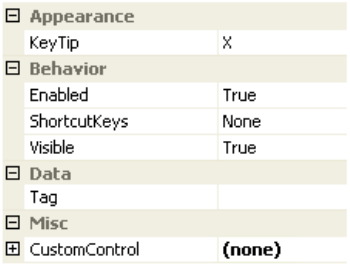


Figure 1 - Group Item CustomControl Properties

KeyTip

When *KeyTips* are displayed this property defines the *KeyTip* for the custom control instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

Enabled

Used to define if the custom control is enabled or disabled at runtime.

ShortcutKeys

Define this property if you would like your application to have a shortcut key combination that sets focus to the custom control.

Visible

Use this property to specify if the custom control should be visible at runtime.

CustomControl

Assign to this property a reference to the actual control instance you would like displayed inside the ribbon. See the *Ribbon Custom Controls* sample for an example of this in action and also the source code of that sample to see the simple process of creating and assigning instances during the *Form* load event.

Tag

Associate application specific information with the object instance by using this property.

Group Item DateTimePicker

Use this element to add a date time picker control to the ribbon. The set of available properties is shown in Figure 1.

Appearance	
DropDownAlign	Left
Format	Long
KeyTip	X
ShowCheckBox	False
ShowUpDown	False
UseMnemonic	True
Value	11/03/2009 11:36 AM
ValueNullable	11/03/2009 11:36:20 AM
Behavior	
Checked	True
ContextMenuStrip	(none)
CustomFormat	
Enabled	True
KryptonContextMenu	(none)
MaxDate	31/12/9998
MinDate	1/01/1753
ShortcutKeys	None
Visible	True
Data	
Tag	
Layout	
MaximumSize	180, 0
MinimumSize	180, 0
MonthCalendar	
CalendarAnnuallyBoldedDates	DateTime[] Array
CalendarBoldedDates	DateTime[] Array
CalendarDimensions	1, 1
CalendarFirstDayOfWeek	Default
CalendarMonthlyBoldedDates	DateTime[] Array
CalendarShowToday	True
CalendarShowTodayCircle	True
CalendarShowWeekNumbers	False
CalendarTodayDate	11/03/2009
CalendarTodayText	Today:
Visuals	
AllowButtonSpecToolTips	False
ButtonSpecs	**(Collection)**
Visuals - MonthCalendar	
CalendarDayOfWeekStyle	Calendar Day
CalendarDayStyle	Calendar Day
CalendarHeaderStyle	Calendar

Figure 1 - Group Item DateTimePicker Properties

DropDownAlign
Determines if the drop down month calendar appears aligned to the left or right edge of the date time picker.
Format
Offers predefined ways of formatting the text portion of the control.
KeyTip
Character used to select the control when the user presses ALT to gain keyboard access to the ribbon.
ShowCheckBox
Defaults to *False* and so does not show a check box in the control but when defined allows the user to decide if the value in the control is valid.
ShowUpDown
Defaults to *False* and so shows a drop down button but when defined as *True* will display up/down buttons for modifying contents.
Value
A *DateTime* property that represents the value inside the control.
ValueNullable
A *object* type property that returns *DBNull* when the *Checked* property is *False* but a valid *DateTime* when the *Checked* property is *True*.
Checked
When defined the value of the control is valid, otherwise the value is not valid and drawn disabled.

CustomFormat
If the *Format* property is defined as *Custom* then this property defines how the text is formatted.
CustomNullText
If the *Checked* property is *False* then this text is drawn in the text area unless the property is set to an empty string.
Enabled
Should the month calendar be displayed as enabled and allow interaction with the user. Note that if the *KryptonContextMenu* component has its own *Enabled* property defined as *False* then the item will be disabled regardless of the individual menu item *Enabled* state.
MaxDate, MinDate Place limits on the displayed and selectable date range by using these two properties.

Visible
Define this as *False* if you do not want the element to be displayed.
ShortcutKeys
Define this property if you would like your application to have a shortcut key combination that invokes the date time picker.
Tag
Use the *Tag* to assign your application specific information with the component instance.
MaximumSize MinimumSize
These two properties are very important and allow you to control the width of the date time picker. By default they have the same value giving a fixed width to the control, but you can alter the values so that they specify a valid range of widths. When the control is positioned it is asked for its preferred size which is then constrained by these two properties.

CalendarAnnuallyBoldedDates, CalendarMonthlyBoldedDates, CalendarBoldedDates
Use these collections to specify which dates should be displayed in a bold state.

CalendarDimensions
Determines the number of months shown as a grid.

CalendarFirstDayOfWeek Specify which day of week is used as the first displayed column within the month.

CalendarShowToday, CalendarShowTodayCircle
Determines if today's date is displayed in the bottom caption of the control and if the day that represents today's date has a highlighted border (circle).

CalendarShowWeekNumbers
When defined this property will show week numbers in the row header of each displayed week of values.

CalendarTodayDate
Defaults to the current date when the control is created but can be set to define any date as the today date.

CalendarTodayText
Text used to caption the today's date in the bottom caption area.

AllowButtonSpecTooltips
By default the control will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.
ButtonSpecs
Use this collection property to define any number of extra buttons that you would like to appear at the near or far edges of the control.

CalendarDayOfWeekStyle, CalendarDayStyle, CalendarHeaderStyle
Properties used to define the palette styles used to draw various elements of the control.

Group Item Gallery

Use a gallery instance to present the user with a set of images that they can single select from. The user can use the up and down buttons on the side of the gallery to scroll additional rows of images into view. They can also press a drop down button to show a context menu containing the full list of images. If space is constrained then the gallery is replaced with a large button, pressing the large button causes the gallery drop down menu to be shown. Figure 1 shows the list of all properties exposed by the group gallery item.

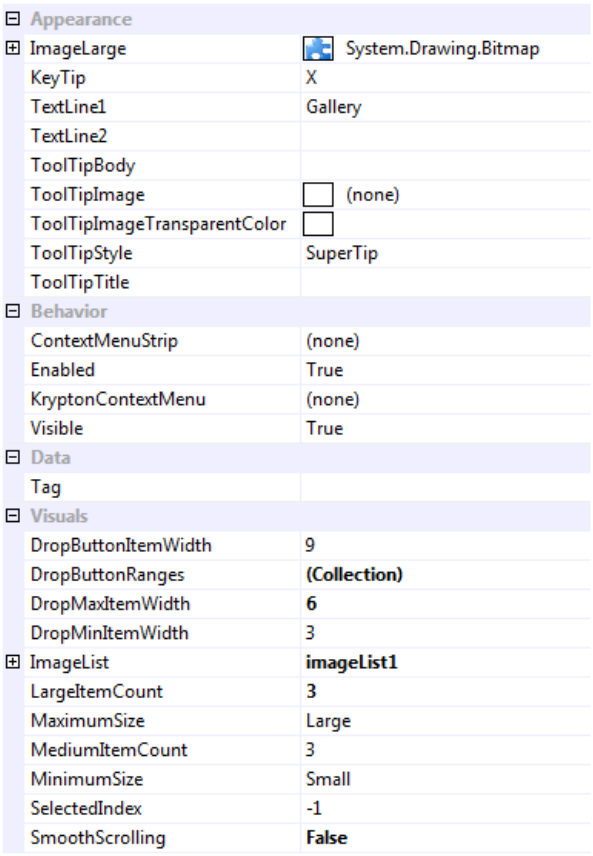


Figure 1 - Group Item Gallery Properties

ImageLarge
This property must always have a valid image defined. If you assign *null* to the properties then it will automatically revert to a default blue jigsaw piece image instead. Once the gallery is in the small setting it replaces the gallery appearance with a button appearance and this image is used in the button appearance.

KeyTip
When *KeyTips* are displayed this property defines the KeyTip for the gallery instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

TextLine1 TextLine2
When showing as a button the *TextLine1* and *TextLine2* strings are shown on two separate lines underneath the *ImageLarge*.
ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle
When the user hovers the mouse over the button appearance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.

ContextMenuStrip
When the user right clicks the gallery this property is used if defined and the *KryptonContextMenu* property is null.

Enabled
Used to define if the gallery is enabled or disabled at runtime.

KryptonContextMenu
When the user right clicks the gallery this context menu is used. If not defined then the *ContextMenuStrip* is used instead.

Visible
Use this property to specify if the gallery should be visible at runtime.

DropButtonItemWidth
When the gallery is displayed as a button and the user presses the button it shows a drop down context menu with all the gallery images available for selection. Use this setting to define the number of items per horizontal line in that drop down menu. Note this property only applies when the gallery has been reduce to show as a button.
DropButtonRanges
If this collection is empty then all the gallery images are shown in one large group within the drop down menu. In order to change the grouping you add entries to this collection that define a header title and the range of items it should contain. This allows you to split the display images into groups that are titled.

DropMaxItemWidth DropMinItemWidth
By default when the drop down menu is shown for the gallery the number of items per horizontal line will match the current number of items showing in the gallery itself. So if the gallery is showing a width of 4 items then the drop down menu will show items with 4 per line. Use these two properties to define min/max values for the drop down items per line.

ImageList
Reference to image list that contains all the display images.
LargeItemCount MediumImageCount
The large item count defines the maximum number of items to show in the gallery. This therefore limits the maximum width of the gallery to that needed to show the large item count number of items. As space becomes constrained it will reduce the width of the gallery by until the number of showing items matches the medium item count. Any attempt to reduce the gallery size will then remove the gallery and replace it with a button appearance instead.
MaximumSize
MinimumSize
Use these two properties to define the maximum and minimum group sizes allowed. Possible values are large, medium and small. The ribbon will always try to show the gallery at the largest possible size allowed by the available size. If the gallery is reduced to the small setting then the gallery is replaced with a button instead.

SelectedIndex
Index of the currently selected image for the gallery.
SmoothScrolling
Determines if scrolling occurs as a smooth animation or if instead an immediate jump is made to the destination.

Events

GalleryDropDown
A cancelable event than allows you to customize the drop down context menu before it is displayed.

TrackImage
As the user tracks over different images this event fires so you can provide instance feedback on the effect this would have if selected.

SelectedIndexChanged
Occurs when the *SelectedIndex* property changes.

ImageListChanged
Fired when the value of the ImageList property changes.

Group Item NumericUpDown

Use this element to add a numeric up down control to the ribbon. The set of available properties is shown in Figure 1.

Appearance	
Hexadecimal	False
KeyTip	X
TextAlign	Left
UpDownAlign	Right
Value	0
Behavior	
ContextMenuStrip	(none)
Enabled	True
InterceptArrowKeys	True
ReadOnly	False
ShortcutKeys	None
Visible	True
Data	
DecimalPlaces	0
Increment	1
Maximum	100
Minimum	0
Tag	
ThousandsSeparator	False
Layout	
MaximumSize	121, 0
MinimumSize	121, 0
Visuals	
AllowButtonSpecToolTips	False
ButtonSpecs	**(Collection)**

Figure 1 - Group Item NumericUpDown Properties

Hexadecimal
Determines if the numeric is display in hex (base 16) format or in the usual decimal (base 10) format.

KeyTip
When *KeyTips* are displayed this property defines the *KeyTip* for the custom control instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

TextAlign
Defines the alignment of the numeric value which can be either left, centered or the right.

UpDownAlign
Should the up-down spin buttons be placed to the right or left of the edit box.

Value
The current decimal value as entered by the user into the edit box.

ContextMenuStrip
Reference to menu strip associated with the control instance.

Enabled
Used to define if the text box is enabled or disabled at runtime.

InterceptArrowKeys
True if you want the numeric control to use the up and down arrow keys to modify the edit value.

ReadOnly
Should the value be presented as read only for the user.

ShortcutKeys
Define this property if you would like your application to have a shortcut key combination that sets focus to the numeric.

Visible
Use this property to specify if the text box should be visible at runtime.

DecimalPlaces
How many digits should be displayed after the decimal place.

Increment
The decimal value to add or subtract when the user presses the spin buttons or up and down arrow keys.

Maximum
The upper limit allowed for the value of the numeric.

Minimum
The lower limit allowed for the value of the numeric.

Tag
Associate application specific information with the object instance by using this property.

ThousandsSeparator
When defined the culture specific thousands separator will be shown between each group of three digits.

MaximumSize MinimumSize
These two properties are very important and allow you to control the width of the combo box. By default they have the same value giving a fixed width to the control, but you can alter the values so that they specify a valid range of widths. When the control is positioned it is asked for its preferred size which is then constrained by these two properties.

AllowButtonSpecTooltips
By default the text box will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

ButtonSpecs
You can add extra buttons to the text box by modifying the *ButtonSpecs* collection. Each *ButtonSpec* entry in the collection describes a single button for display. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances.

Hidden Property
In order to gain access to the underlying *KryptonNumericUpDown* instance you can use the hidden property called *NumericUpDown*. This is not displayed at design time and not persisted but can be accessed via code at runtime. This is useful if you need to access a property that is not exposed via this ribbon element.

Group Item RadioButton

You can add a radio button to either the group triple or the group lines container. Use the *Checked* property to define the checked state of the radio button. Figure 1 shows the list of all properties exposed by the group radio button item.

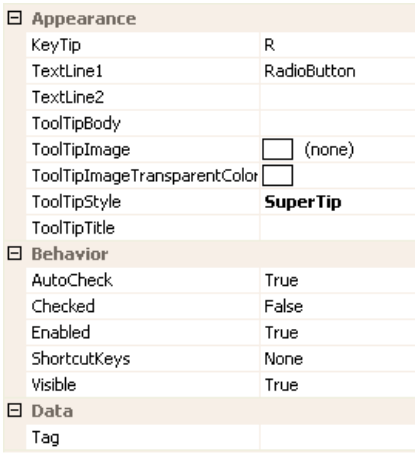


Figure 1 - Group Item RadioButton Properties

KeyTip

When *KeyTips* are displayed this property defines the *KeyTip* for the radio button instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

TextLine1 TextLine2

When the radio button is inside a container that displays it the full height of the group content area the *TextLine1* and *TextLine2* strings are shown on two separate lines underneath the radio button image. In all other cases the *TextLine1* and *TextLine2* are concatenated together with a space between them for showing horizontally after the radio button image.

ToolTipBody ToolTipImage
ToolTipImageTransparentColor ToolTipStyle
ToolTipTitle

When the user hovers the mouse over the radio button instance you can use these properties to define the tool tip that will be displayed. Use *ToolTipTitle* and *ToolTipBody* to define the two text strings for display and *ToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *ToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *ToolTipStyle* property.

AutoCheck

When defined as *True* this property will ensure that the radio but is automatically reset to false when any other radio button inside the same ribbon group becomes checked. This allows a group of radio button instances to act as a group where only a single radio button is checked at any time. It also allows the user to alter the checked state by clicking the control.

Checked

A boolean property this indicates if the radio button is currently checked or not.

Enabled

Used to define if the radio button is enabled or disabled at runtime.

ShortcutKeys

Define this property if you would like your application to have a shortcut key combination that invokes the radio button.

Visible

Use this property to specify if the radio button should be visible at runtime.

Tag

Associate application specific information with the object instance by using this property.

Events

Click

Occurs when the radio button has been pressed.

CheckedChanged

Occurs when the *Checked* property changes value.

Group Item RichTextBox

Use this element to add a rich text box control to the ribbon. The set of available properties is shown in Figure 1.

Appearance	
DropDownStyle	DropDown
KeyTip	X
Text	
Behavior	
ContextMenuStrip	(none)
DropDownHeight	106
DropDownWidth	143
Enabled	True
ItemHeight	**13**
MaxDropDownItems	8
MaxLength	0
ShortcutKeys	None
Sorted	False
Visible	True
Data	
DataSource	(none)
DisplayMember	(none)
Items	**(Collection)**
Tag	
ValueMember	
Layout	
MaximumSize	121, 0
MinimumSize	121, 0
Misc	
AutoCompleteCustomSourc	**(Collection)**
AutoCompleteMode	None
AutoCompleteSource	None
FormatString	
FormattingEnabled	**False**
Visuals	
AllowButtonSpecToolTips	False
ButtonSpecs	**(Collection)**

Figure 1 - Group Item RichTextBox Properties

KeyTip

When *KeyTips* are displayed this property defines the *KeyTip* for the custom control instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

Enabled

Used to define if the rich text box is enabled or disabled at runtime.

ShortcutKeys

Define this property if you would like your application to have a shortcut key combination that sets focus to the rich text box.

Visible

Use this property to specify if the rich text box should be visible at runtime.

MaximumSize MinimumSize

These two properties are very important and allow you to control the width of the combo box. By default they have the same value giving a fixed width to the control, but you can alter the values so that they specify a valid range of widths. When the control is positioned it is asked for its preferred size which is then constrained by these two properties.

ButtonSpecs

You can add extra buttons to the rich text box by modifying the *ButtonSpecs* collection. Each *ButtonSpec* entry in the collection describes a single button for display. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances.

AllowButtonSpecTooltips

By default the rich text box will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

Tag

Associate application specific information with the object instance by using this property.

Other Properties

All the other properties displayed in Figure 5 are just properties from the underlying *KryptonRichTextBox* instance and exposed via this element for ease of access. For more information about these properties see the standard windows documentation for the *RichTextBox* control.

Hidden Property

In order to gain access to the underlying *KryptonRichTextBox* instance you can use the hidden property called *RichTextBox*. This is not displayed at design time and not persisted but can be accessed via code at runtime. This is useful if you need to access a property that is not exposed via this ribbon element.

Group Item TextBox

Use this element to add a text box control to the ribbon. The set of available properties is shown in Figure 1.

Appearance	
DropDownStyle	DropDown
KeyTip	X
Text	
Behavior	
ContextMenuStrip	(none)
DropDownHeight	106
DropDownWidth	143
Enabled	True
ItemHeight	**13**
MaxDropDownItems	8
MaxLength	0
ShortcutKeys	None
Sorted	False
Visible	True
Data	
DataSource	(none)
DisplayMember	(none)
Items	**(Collection)**
Tag	
ValueMember	
Layout	
MaximumSize	121, 0
MinimumSize	121, 0
Misc	
AutoCompleteCustomSourc	**(Collection)**
AutoCompleteMode	None
AutoCompleteSource	None
FormatString	
FormattingEnabled	**False**
Visuals	
AllowButtonSpecToolTips	False
ButtonSpecs	**(Collection)**

Figure 1 - Group Item TextBox Properties

KeyTip

When *KeyTips* are displayed this property defines the *KeyTip* for the custom control instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

Enabled

Used to define if the text box is enabled or disabled at runtime.

ShortcutKeys

Define this property if you would like your application to have a shortcut key combination that sets focus to the text box

Visible

Use this property to specify if the text box should be visible at runtime.

MaximumSize MinimumSize

These two properties are very important and allow you to control the width of the combo box. By default they have the same value giving a fixed width to the control, but you can alter the values so that they specify a valid range of widths. When the control is positioned it is asked for its preferred size which is then constrained by these two properties.

ButtonSpecs

You can add extra buttons to the text box by modifying the *ButtonSpecs* collection. Each *ButtonSpec* entry in the collection describes a single button for display. You can use the *ButtonSpec* to control all aspects of the displayed button including visibility, edge, image, text and more. At design time use the collection editor for the *ButtonSpecs* property in order to modify the collection and modify individual *ButtonSpec* instances.

AllowButtonSpecTooltips

By default the text box will not show any tool tips when you hover the mouse over the user defined button specifications. If you set this boolean property to *True* then it will turn on tool tips for those button specs. Use the *ButtonSpec.TooltipText* property in order to define the string you would like to appear inside the displayed tool tip.

Tag

Associate application specific information with the object instance by using this property.

Other Properties

All the other properties displayed in Figure 5 are just properties from the underlying *KryptonComboBox* instance and exposed via this element for ease of access. For more information about these properties see the standard windows documentation for the *ComboBox* control.

Hidden Property

In order to gain access to the underlying *KryptonTextBox* instance you can use the hidden property called *TextBox*. This is not displayed at design time and not persisted but can be accessed via code at runtime. This is useful if you need to access a property that is not exposed via this ribbon element.

Group Item TrackBar

Use this element to add a track bar control to the ribbon. The set of available properties is shown in Figure 1.

Appearance	
KeyTip	T
Orientation	Horizontal
TickFrequency	1
TickStyle	None
TrackBarSize	Medium
VolumeControl	False
Behavior	
ContextMenuStrip	(none)
Enabled	True
KryptonContextMenu	(none)
LargeChange	5
Maximum	**8**
Minimum	0
SmallChange	1
Value	0
Visible	True
Data	
(ApplicationSettings)	
Tag	
Design	
(Name)	**kryptonRibbonGroupTrackBar1**
GenerateMember	True
Modifiers	Private
Layout	
MaximumLength	**55**
MinimumLength	**55**

Figure 1 - Group Item TrackBar Properties

KeyTip

When *KeyTips* are displayed this property defines the *KeyTip* for the custom control instance. You should ensure that all items inside a tab have unique *KeyTip* values so that the user can always select items using keyboard access.

Orientation

Change the direction of the track bar by switching this property from the default of *Horizontal* to *Vertical*.

TickFrequency

A tick mark is drawn depending on the frequency value defined. A frequency value of 1 draws a tick mark for each value between the *Minimum* and *Maximum*, a frequency of 5 would only draw a tick mark for every 5th value between those value limits.

TickStyle

You can prevent any ticks marks being drawn by setting this value to None. Use the other options allows ticks to be shown either to one side of the track or to both.

TrackBarSize

The track bar can be shown in either *Small*, *Medium* or *Large* variations. Choose the variation that gives the appearance required for your application.

VolumeControl

Setting this to true will turn the control into drawing as a volume control with the track looking like a wedge.

ContextMenuStrip

Assign a standard windows *ContextMenuStrip* instance to this property for display on right clicking the control.

Enabled

Used to define if the track bar is enabled or disabled at runtime.

KryptonContextMenu

Assign a *KryptonContextMenu* instance to this property for display on right clicking the control.

LargeChange

This value is added or subtracted from the *Value* when the user clicks to the side of the position indicator.

Maximum

Defines the maximum allows value of the *Value* property.

Minimum
Defines the minimum allows value of the *Value* property.

SmallChange
Use this property to specify if the text box should be visible at runtime.

Value
T

Visible
Use this property to specify if the text box should be visible at runtime.

Tag
Associate application specific information with the object instance by using this property.

MinimumLength
The minimum pixel length of the control. When the Orientation is Horizontal the length is the Width otherwise the Height of the control.

MaximumLength
The maximum pixel length of the control. When the Orientation is Horizontal the length is the Width otherwise the Height of the control.

Ribbon Contextual Tabs

To implement contextual tabs you need to understand the relationship between the ribbon *SelectedContext* property, the ribbon *RibbonContexts* collection and the *ContextName* property defined for each ribbon tab instance.

The *RibbonContexts* property on the ribbon control is a collection property of individual *KryptonRibbonContext* instances. Each *KryptonRibbonContext* instance defines one possible context. You should create as many instances as different contexts you need for the ribbon. For example, if your application has the ability to select pictures and charts then you would create two contexts. Your first context would have a title of *Picture* and the second a title of *Chart*.

If the user selected a chart and a picture at the same time then you can display both contexts at the same time by setting the appropriate values into the ribbon *SelectedContext* property. The *SelectedContext* property is a comma separated list of contexts your would like to display. So if you have two contexts defined with names of *Picture* and *Chart* the valid combinations for the *SelectedContext* property would be:-

- Picture
- Chart
- Picture,Chart
- Chart,Picture

The order of the context names is important. The ribbon will show the context groupings of tabs in the same order that they are defined in the SelectedContext. So with a value of 'Chart,Picture' the chart context tabs would be shown before the picture context tabs.

To associate a ribbon tab so it is displayed only in the context of a particular context you need to set the *KryptonRibbonTab.ContextName* property. The value should match the same name as in the *KryptonRibbonContext.ContextName* instance of interest.

KryptonRibbonContext Properties

Figure 1 shows the properties available for a ribbon context instance.

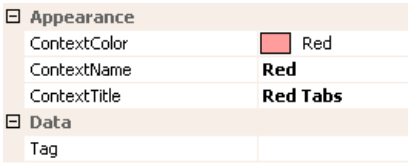


Figure 1 - Ribbon Context Properties

ContextColor

When the context is being displayed this is the color used to draw the context indication area at the top of the ribbon control. If the ribbon control is integrated into the caption area of the form then this color will be seen in the caption area.

ContextName

The *ContextName* is used to associate the individual tabs with the context instance. To associate a tab with this context you should set the *KryptonRibbonTab.ContextName* to the same value as this property.

ContextTitle

This is the title text that is drawn over the *ContextColor* background color. You should use a short string to reduce the chances of it being truncated and a string that reflects the context usage, good examples would be *Picture*, *Table* and *Chart*.

Tag

Associate application specific information with the object instance by using this property.

Ribbon Application Button

The application button is the large round button that appears at the top left of the ribbon control. You cannot prevent the display of the button and pressing it should always result in the display of a context menu. When the ribbon control is integrated into the custom chrome of the owning *Form* then the application button should be presented half in the *Form* caption area and half in the ribbon control itself. When the ribbon is not integrated then an extra header area is shown at the top of the ribbon control that allows the application button to be completely shown.

Application Button Properties
You can see in Figure 1 the ribbon properties relating to the application button.

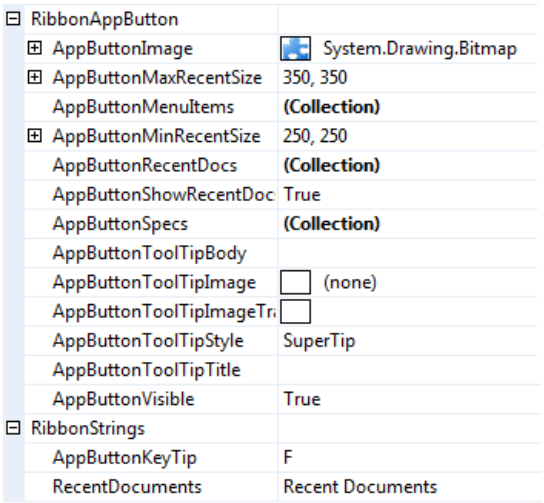


Figure 1 - Application Button Properties

AppButtonImage
Use this property to define the image that is displayed in the center of the application button. You should always assign an *Image* of size 24 x 24 pixel as the ribbon will always stretch the provided image to be drawn in that fixed size. Note that this property cannot be assigned *null* and so if you cannot supply a value of your own it will default to the blue jigsaw piece seen in the sample applications.

AppButtonMenuItems
Defines the hierarchy of context menu items that are displayed on the left side of the application menu. Note that the first level of sub menus are displayed as a fixed size that overlap the recent documents area. Hence it is important to have a reasonably large recent documents area so that all the first level sub menus will be fully displayed. The exception to this rule is when the recent documents area is turned off via the *AppButtonShowRecentDocs* property, in which case the first level sub menus are shown sized according to content.

AppButtonMinRecentSize
The recent documents area of the application menu honors a minimum size as specified by this property. This ensures that when you only have a small number of entries you still reserve a reasonable amount of display space. It is also essential when you have large first sub menus defined. The first level of sub menus are always displayed as a fixed size that overlay the recent documents area. Therefore you would need to use this property to ensure that the sub menu always has enough size that the sub menu can be fully displayed.

AppButtonMaxRecentSize
Adding recent document entries that have large text values, such as a full path filename, it can quickly make the application menu very wide. Use this property to limit the width of the recent documents area so it does not become unmanageable.

AppButtonRecentDocs
Each entry in this collection will appear as an option inside the recent documents section of the application menu.

AppButtonShowRecentDocs
The recent documents section of the application menu is optional. If you set this property to *False* then this section will not be displayed. This is a useful if your application does not support the concept of recent documents and also has no need to reuse the section or another purpose.

AppButtonSpecs
You can use this collection to define additional buttons that appear at the bottom of the application menu. Typically you would use this feature to add an *Exit* or *Options* button in the same manner as the *Office 2007* applications. They should be used when it does not make sense to add the same capability to the context menu portion of the application menu.

AppButtonToolTipBody AppButtonToolTipImage
AppButtonToolTipImageTransparentColor AppButtonToolTipStyle

AppButtonToolTipTitle
When the user hovers the mouse over the application button instance you can use these properties to define the tool tip that will be displayed. Use *AppButtonToolTipTitle* and *AppButtonToolTipBody* to define the two text strings for display and *AppButtonToolTipImage* for the associated image. If you image contains a color that you would like to be treated as transparent then set the *AppButtonToolTipImageTransparentColor*. For example, many bitmaps use magenta as the color to become transparent. To control how the text and image are displayed in the tool tip you can use the *AppButtonToolTipStyle* property.

AppButtonVisible
Determines if the application button is displayed with the ribbon.

AppButtonKeyTip
When the user enter *KeyTips* mode by using the *ALT* key or *F10* this property is used to decide on the *KeyTip* text to be used for the application button. By default the single letter *F* is used because the most likely use of the application button is to show a *File* context menu. This property is localizable so you can change the value on a per culture basis.

RecentDocuments
This property defines the string that is displayed as a title for the recent documents section of the application menu. You can change the title if you require to repurpose the use of that menu section. This property is localizable so you can change the value on a per culture basis.

Ribbon Quick Access Toolbar

Use the quick access toolbar to allow one click access to common functionality in your application. The quick access toolbar looks like a traditional toolbar and can be placed either above or below the main ribbon control area. If there are too many entries in the toolbar to show them all at once then an extra overflow button is shown so that you can click and see a pop up with the additional entries. The user can use the customize button to change the visible state of the individual toolbar entries.

Quick Access Toolbar Properties
You can see in Figure 1 the ribbon properties relating to the quick access toolbar.

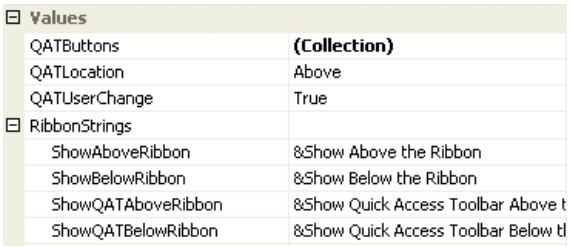


Figure 1 - Quick Access Toolbar Properties

QATButtons
This is a collection property that defines each of the quick access toolbar button entries. To add, remove or modify new entries click this property and use the collection editor. Note that some of the changes you make inside the collection editor will not update the ribbon control until you have exited the collection editor.

QATLocation
By default the quick access toolbar will be positioned above the ribbon and next to the application button. You can alter this property to *QATLocation.Below*in order to have the toolbar placed in a bar on its own below the main ribbon area. If you like to completely remove it from being displayed then use the *QATLocation.Hidden* setting.

QATUserChange
When the user clicks the customize button a context menu is shown that by default shows a list of all the quick access toolbar entries along with the visible state of those entries. Visible entries are checked and hidden entries not checked. At runtime the user can select the entries in order to toggle the visibility. If you need to prevent the user from altering the visible state then set this property to *False*. This will cause the customize button to show a context menu that does not show any toolbar entries.

ShowAboveRibbon ShowBelowRibbon
ShowQATAboveRibbon ShowQATBelowRibbon
There are two different context menus for the ribbon control that allow the user to switch the location of the quick access toolbar at runtime between above and below. These four string properties define the text that is shown for the context menu items. The properties are localizable so you can update the text with a string that is appropriate for the selected culture.

Ribbon KeyTips & Keyboard Access

KeyTips are a quick way to invoke ribbon functionality using only the keyboard. Use the *ALT* or *F10* keys to enter *KeyTips* mode and you will notice that several elements of the ribbon are overlaid with small tool tip style windows. If you press the key indicated by any of the *KeyTips* then the matching element is invoked. For example, pressing the *KeyTip* for the application button will cause the button to be invoked and the associated context menu to be displayed. Pressing the *KeyTip* displayed over a tab will navigate into the tab and have the *KeyTips* for the groups inside the tab displayed. You can use the *ESC* key at any time to cancel the last action.

In order to enter keyboard access mode you first enter *KeyTips* mode by using the *ALT* or *F10* keys. Then instead of pressing a *KeyTip* you use either the *TAB* key or one of the *LEFT*/*RIGHT*/*UP*/*DOWN* navigation keys. This causes a switch into keyboard access mode which removes the *KeyTips* from display and instead highlights the item that has the keyboard focus. Now use the *TAB* key or one of the *LEFT*/*RIGHT*/*UP*/*DOWN* navigation keys to move around the various elements that make up the ribbon. When on an element of interest such as a group button you can use the *SPACE*/*ENTER*/*DOWN* keys to invoke an appropriate action. Use the *ESC* key at any time to exit keyboard mode.

Keyboard Properties
You can see in Figure 1 the ribbon properties relating to keyboard access.

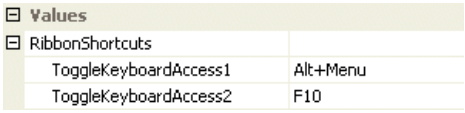


Figure 1 - Keyboard Properties

ToggleKeyboardAccess1 ToggleKeyboardAccess2
You can modify the two keyboard combinations that switch into keyboard access modes by using these two properties. By default these values match those used by the *Microsoft Office 2007* applications but you can modify these to match your application requirements if needed.

Ribbon ButtonSpecs

Use *ButtonSpec* definitions when you need to add additional buttons to the left or right side of the ribbon tabs. Figure 1 shows an example of the ribbon with a *ButtonSpec* defined for the *Near* side with the text *Spec1* and a *ButtonSpec* defined for the *Far* side called *Spec2*. You are not limited to just using text and can use any combination of text and *Image* properties. For a more detailed explanation of all the different *ButtonSpec* properties see this link.

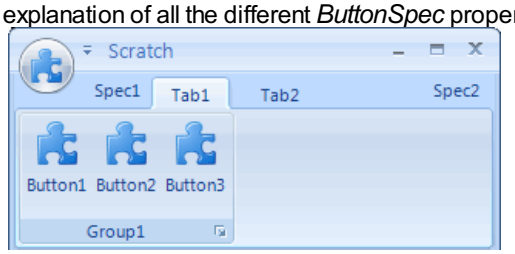


Figure 1 - Ribbon with ButtonSpec instances

ButtonSpec Property

You can see in Figure 1 the ribbon property relating to ButtonSpec instances.

Figure 1 - ButtonSpec Property

ButtonSpecs

The *ButtonSpecs* property is a collection of individual *ButtonSpec* instances. To add, remove or modify the entries you should click the property and then use the presented ellipses button to open the standard collection editor. Note that some changes you make inside the collection editor will not update the ribbon control until you leave by using the *OK* button.

Ribbon Control Events

Applicable Events: ApplicationButtonOpening
ApplicationButtonOpened ApplicationButtonClosing
ApplicationButtonClosed SelectedContextChanged
SelectedTabChanged ShowRibbonContextMenu

ShowQATCustomizeMenu

ApplicationButtonOpening
Generated when the application button is about to show the application menu. This event can be cancelled so if you decide to prevent the showing of the context menu you can simply set the *CancelEventArgs.Cancel* property to *True* and return from your event handler. Alternatively you can use the event as a chance to modify the entires that will appear. So you could update the visible and enabled state of the context menu items or alter the list of recent documents to reflect recent changes.

ApplicationButtonOpened
Generated when the application menu has been created and displayed to the user.

ApplicationButtonClosing
Generated when the application menu makes a request to be dismissed. This event can be cancelled and so you can prevent the menu from being removed. One of the parameters is a value indicating the reason for the close request. So you can discover if the request is the result of an item being click, the escape key being pressed etc.

ApplicationButtonClosed
Generated once the application menu has been removed from display, you cannot cancel this event.

SelectedContextChanged
Whenever the value of the *SelectedContext* property on the *Ribbon* changes this event is generated.

SelectedTabChanged
A change in the *SelectedTab* of the *Ribbon* will generate the *SelectedTabChanged* event. This can occur because the user has clicked a tab and so initiated the change in selection. It can also happen if you change a tab property such as the *Visible* state of the currently selected tab, thus causing a different tab to become selected instead. Adding and removing tabs can potentially change the tab selection along with a change in the *SelectedContext* property.

ShowRibbonContextMenu
When the user right clicks the tabs area of the *Ribbon* this event is generated. This allows you to modify the contents of the context menu about to be shown to the user. You might want to add additional entries that are relevant to your specific application needs.

ShowQATCustomizeMenu
On the right hand side of the quick access toolbar is a small button that is used to customize the display of the quick access toolbar. Just before this customization menu is displayed this event is fired. You can cancel the event to prevent the menu appearing or you can modify the menu contents in order to add new menu options.

Ribbon Item Events

Applicable Events: KryptonRibbonGroup.DialogBoxLauncherClick
KryptonRibbonGroupButton.Click KryptonRibbonGroupButton.DropDown
KryptonRibbonGroupClusterButton.Click KryptonRibbonGroupClusterButton.DropDown
KryptonRibbonQATButton.Click

KryptonRibbonGroup

DialogBoxLauncherClick
When a *KryptonRibbonGroup* is showing the dialog launcher button, which can be seen on the bottom right corner of the group title area, the user can press the button in order to generate the event. The *Ribbon* control performs no other action when this button is pressed other than to generate the event and so if you require a dialog box or other user feedback to occur you must implement it inside the event handler.

KryptonRibbonGroupButton
KryptonRibbonGroupClusterButton

Click
A *Click* event is generate only for the button types of *Push* and *Check*.

DropDown
A *DropDown* event is generated only for button types of *DropDown* and *Split*. The event is generated with a *RibbonContextMenuArgs* instance which is used to provide a reference to the *ContextMenuStrip* associated with the button. Note that you must associate a *ContextMenuStrip* with the button yourself using either code or the design time environment, otherwise the value will be null. You can set the *RibbonContextMenuArgs.Cancel* property to *False* if you would like to prevent the *Ribbon* from displaying the context menu strip once the event handler has finished.

KryptonRibbonQATButton

Click
A *Click* event is generate when the user presses the quick access toolbar button.

KryptonGallery

Use a the gallery control to present the user with a set of images that they can single select from. The user can use the up and down buttons on the side of the gallery to scroll additional rows of images into view. They can also press a drop down button to show a context menu containing the full list of images.

Gallery Properties
You can see in Figure 1 the gallery properties that extend the ones already present for a WinForms control.

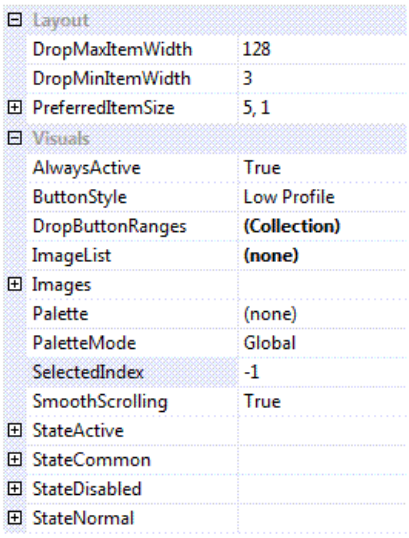


Figure 1 - Gallery specific properties.

DropMaxItemWidth DropMinItemWidth
By default when the drop down menu is shown for the gallery the number of items per horizontal line will match the current number of items showing in the gallery itself. So if the gallery is showing a width of 4 items then the drop down menu will show 4 items per line. However, if the gallery becomes very small it might only be showing one or two items and so showing a drop down menu with just one or two items per line would look inappropriate. Use these two properties to define min/max values for the drop down items per line to ensure the menu always has a reasonable appearance.

PreferredItemSize
If you have set the *AutoSize* property of the control to *True* then you need a way to define the preferred width. Use this property to define the number of images per line and how many lines you would like displayed in the client area of the control. You can see this in action by looking at figure 2. The top instance has the default of 5,1 and the second instance has been changed to 4,2.

Figure 2 - Different PreferredItemSize values.

AlwaysActive
This property is used to indicate if the control should always be in the active state. As a standalone control this default to being True but when used inside a ribbon control instance it is changed to be *False*. This is because you would expect the background to change when the mouse enters the client area inside the ribbon.

ButtonStyle
This style determines how the individual images are displayed. A default of *LowProfile* means that there is no obvious button drawing until you move the mouse over the image or until it becomes selected. But if you prefer another appearance then update the style to any of the other options.

DropButtonRanges
If this collection is empty then all the gallery images are shown in one large group within the drop down menu. In order to change the grouping you add entries to this collection that define a header title and the range of items it should contain. This allows you to split the display images into groups that are titled.

ImageList

Reference to image list that contains all the display images.

Images
This compound property allows you to override the images used in the up/down/drop down buttons that appear on the right hand side of the control. Usually these are inherited from the appropriate palette so that they match the current global style.

Palette
Use this to override the drawing values with those of a specific *KryptonPalette* instance.

PaletteMode
Change this enumeration if you want to select a global palette that comes builtin with the *Krypton Toolkit*.
SelectedIndex
Index of the currently selected image for the gallery.

SmoothScrolling
Determines if scrolling occurs as a smooth animation or if instead an immediate jump is made to the destination.

StateActive StateCommon
StateDisabled StateNormal
All the state compound properties are used to override different elements of the gallery with custom values.

Events

GalleryDropMenu
Fired just before the drop down menu is displayed. You can cancel this event to prevent the drop down menu from appearing or instead customize the contents of the menu to add additional menu items.

TrackingImage
As the user tracks over different images this event fires so you can provide instant feedback to the user about the change that would happen if the image were to be selected. When tracking leaves all the images then you get a value of -1 provided in the event data.

SelectedIndexChanged
Occurs when the *SelectedIndex* property changes value.

Tutorials

Use these simple step by step guides to learn more about using the *Krypton* suite.

Krypton Toolkit Tutorials

Using Krypton in VS2005
Follow these simple steps to add the *Krypton* set of components to the Toolbox in *Visual Studio 2005*.

Using Images with Buttons

See how to quickly achieve the look and feel you want using one of these simple techniques. These apply to *KryptonButton*, *KryptonCheckButton* and *ButtonSpec*.

Embedding Palette Definitions

Embed palette definition XML files into your assembly resources and then load them at runtime. This deployment method reduces the risk of the user deleting your palette definition files.

Click to Enlarge

Multiple Choice Buttons

Learn how to add a group of mutually exclusive buttons to your application. Combine the *KryptonCheckSet* with *KryptonCheckButton* instances in just a few seconds!

Click to Enlarge

Three Pane Application

Starting from a new project you learn how to build a simple three pane application using the *Krypton Toolkit* controls. Use this as a starting point for building up your application.

Click to Enlarge

Expanding HeaderGroups (Splitters)
Allow the user to expand and collapse header groups at runtime. This variation uses split containers so the user can change the relative spacing of the groups.

Click to enlarge

Expanding HeaderGroups (DockStyle)
Allow the user to expand and collapse header groups at runtime. In this variation the groups are docked against the form edges and not a single line of code is needed.

Click to enlarge

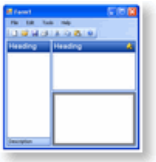
Expanding HeaderGroups (Stack)
Allow the user to expand and collapse header groups at runtime. The header groups use up space from the center group as they expand. Create this scenario without a single line of code.

Click to enlarge

Krypton Navigator Tutorials

User Page Creation

Add Internet Explorer 7 style operation by having a minimum of two pages and clicking the last page creates another page automatically.



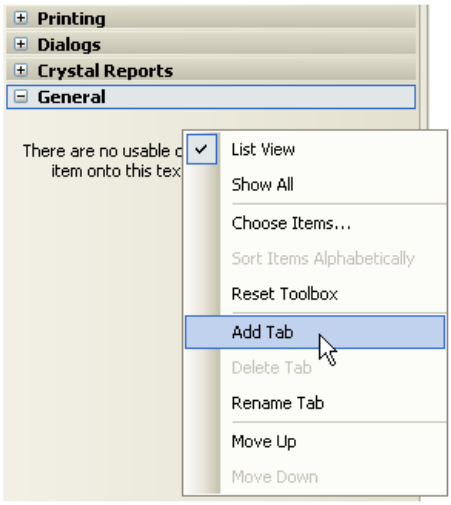
Click to Enlarge

Tutorial – Using Krypton in VS2005

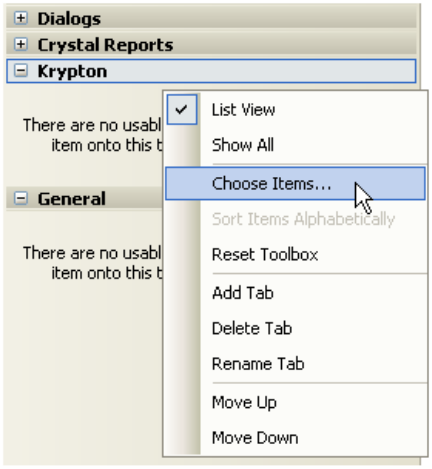
Use this tutorial to to add the *Krypton* set of components to the Toolbox.

1) Right click the toolbox and select 'Add Tab'

Enter the name 'Krypton' for the new tab title.

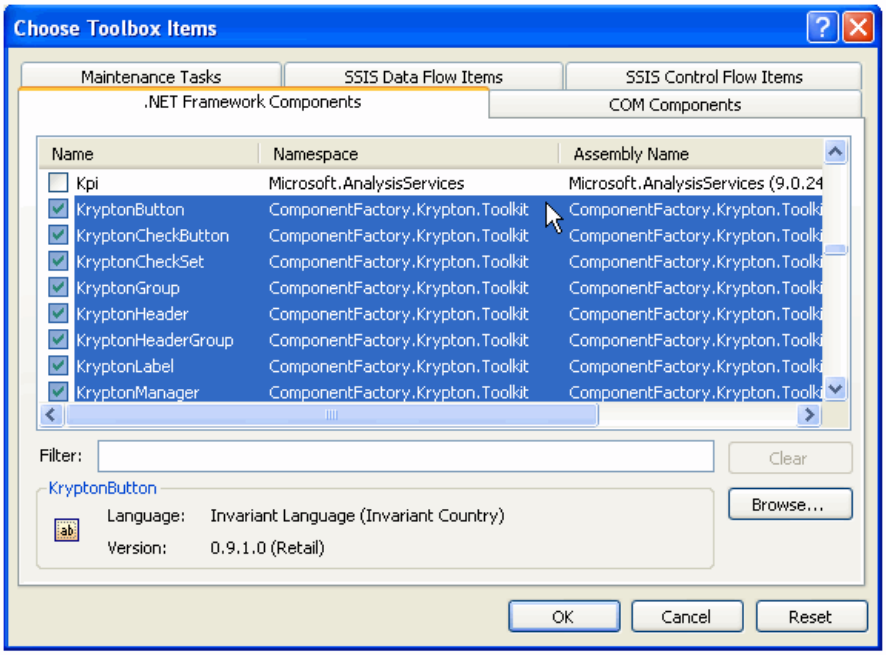


2) Right click inside the new tab and select 'Choose Items...'



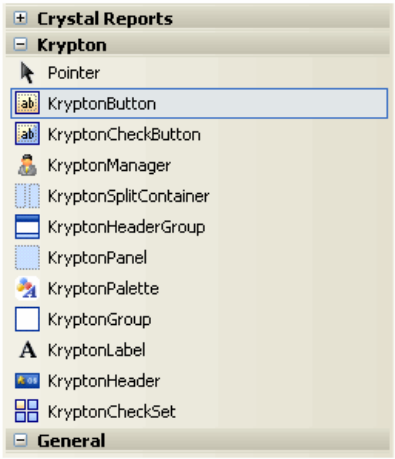
3) Select the Krypton components

The full selection of *Krypton* components will already be present in the dialog box. Just click the *Namespace* column header to sort the list into alphabetical order and then use the scroll bar to bring the *ComponentFactory.Krypton* entries into view. Mark the check box for all the *Krypton* entries so they will all be added to the toolbox.



4) Press 'OK' to add the components to the toolbox

Your new tab should be populated with something like the following.



Tutorial – Using Images with Buttons

Per-State Image Buttons

By using a set of images for all aspects of the appearance you can customize the look and feel in any way you need. Figure 1 shows the value properties for a *KryptonCheckButton* where each of the individual button states has a custom image. The same technique can be applied to the *KryptonButton* control and a *ButtonSpec* definition.

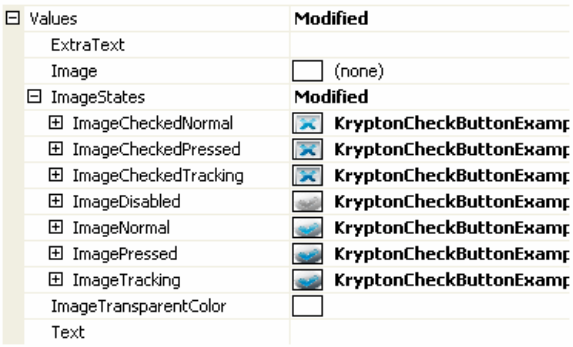


Figure 1 - Using images for each state

For the builtin palettes you will find you need to change three other properties to achieve an image only runtime presentation. First of all you need to prevent the focus rectangle from being drawn when the button has focus. To do this you need to alter the *OverrideFocus -> Content -> DrawFocus* property to *False*. Obviously if you still want the focus rectangle to appear then ignore this step.
Next you need to prevent the button background and border from being drawn. As the entire appearance is represented by the images we do not need a border or background. To turn off the background set the *StateCommon -> Back -> Draw*property to *False*. For the border set to *False* for the *StateCommon -> Border -> Draw*property.

You are now ready to use the button at runtime, figure 2 shows the appearance for all the different button states for the figure 1 setup when the background, border and focus drawing have been turned off.

Figure 2 - Actual check button display

Single Image Buttons

These buttons should look familiar as they are the default type of button presented by the *Professional - Office 2003* builtin palette. They consist of using just a single image that only gets drawn differently in the disabled state. Instead of the image changing when the mouse tracks over and presses down, the background and border are altered . Figure 3 shows how you only need to assign a single image to the properties.

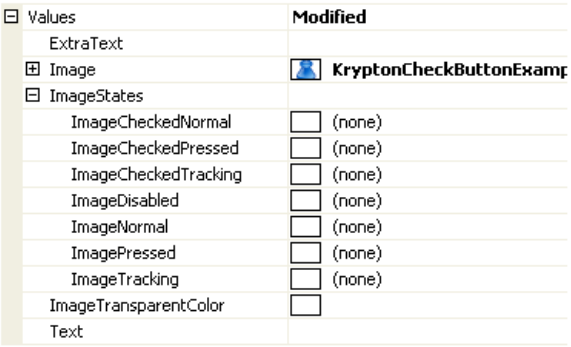


Figure 3 - Single image

You can see in figure 4 how the palette alters the background and border colors to indicate the button state. The only state that draws the image differently is the first one, disabled. If you want to alter the background and border then either create and use a custom palette or alter the control level state definitions.

Figure 4 - Single image button

Color Mapped Image Buttons

Each button state has the ability to remap a single color to a different target color. You can take advantage of this by assigning a single color image to the button and then re mapping that single color in each of the button states. This achieves the advantage of only needing to create and use a single image whilst still being able to alter the image appearance. Figure 5 shows a single black cross image assigned to a button.

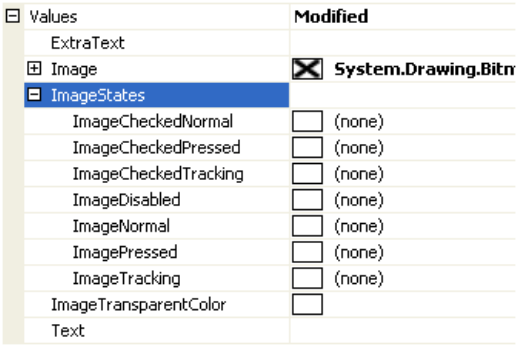


Figure 5 - Single color image

In the mouse tracking mode we want to change the image from black to green. Figure 6 shows the *StateTracking* property called *ImageColorMap* being set the *Black* and the *ImageColorTo* property defined as *Green*.

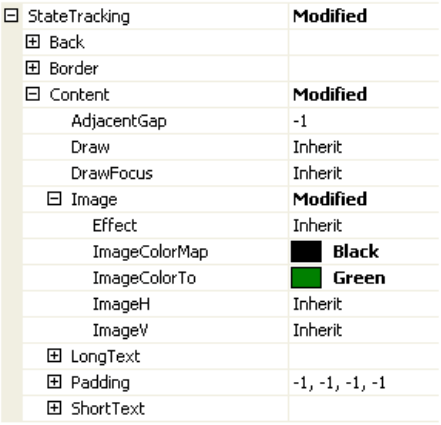


Figure 6 - StateTracking color re mapping

By changing the target color for each of the button states you can see the runtime effect in figure 7 where the background and border drawing have been turned off but the focus rectangle is left to be drawn as normal.

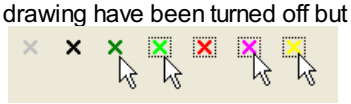


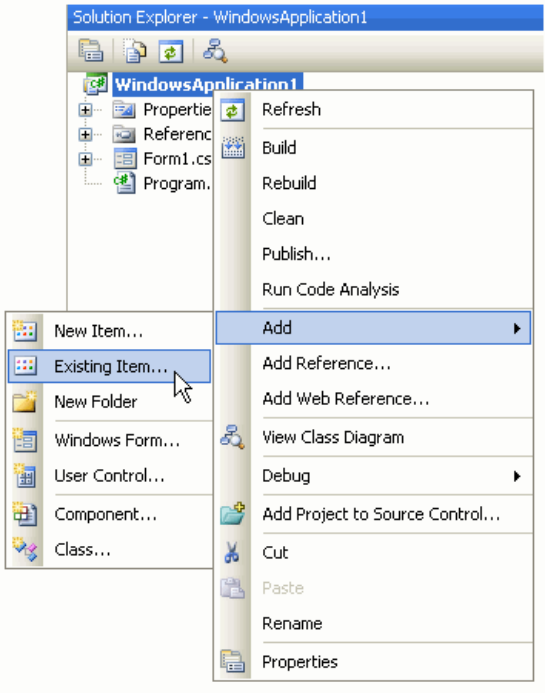
Figure 7 - Color mapped button

Tutorial – Embedding Palette Definitions

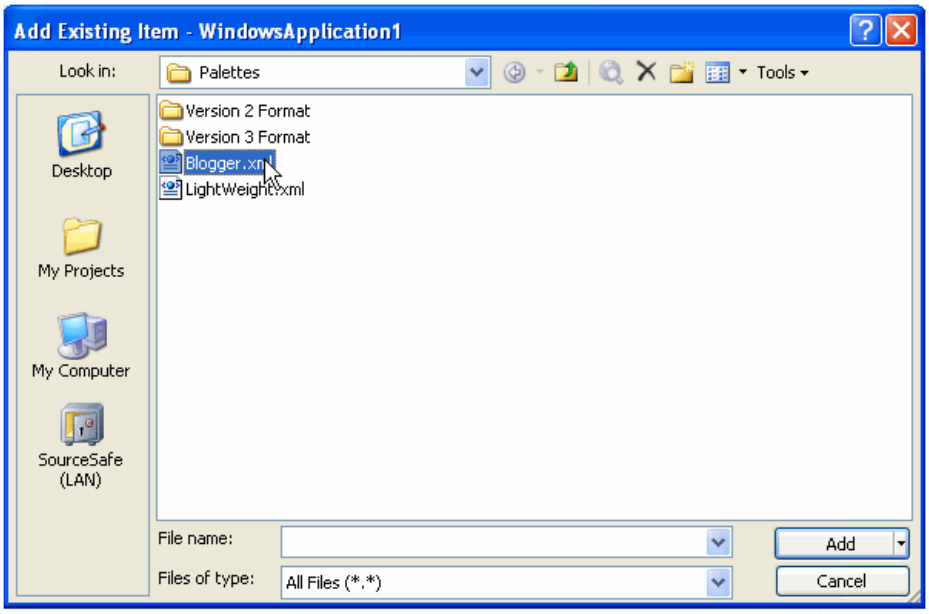
Deploying Palette Definition Files

Rather than distribute palette definition files as separate XML files you might prefer to embed them as resources inside your compiled assembly. This prevents the risk of the user accidentally deleting them and so preventing your application operating as expected. The following steps show how to embed any palette definition as a resource and then how to use just a couple of lines of code to load it at runtime for use.

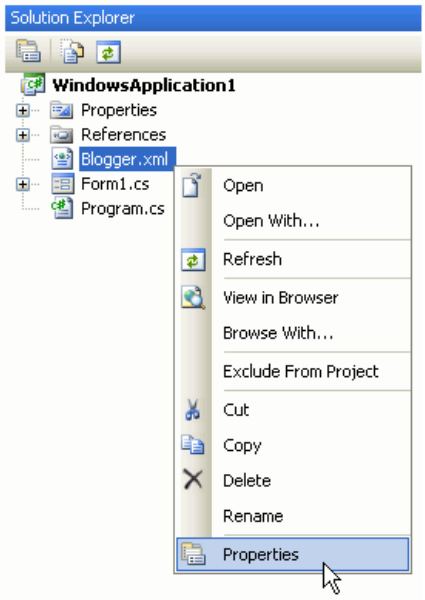
1) Right click your project and select the 'Add -> Existing Item...'
We are using this option in order to add out palette XML file to the project.



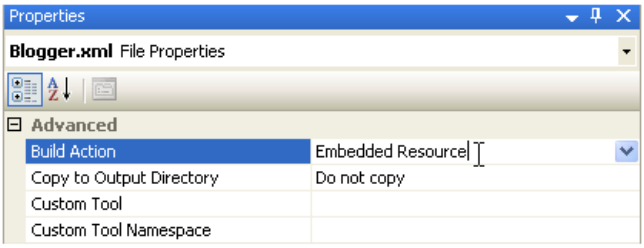
2) Select the palette definition file you want to embed
For this example we are going to choose a palette called, 'Blogger.xml'.



3) Right click the new project item and select 'Properties'



4) Change the 'Build Action' to 'Embedded Resource'
Changing the build action will cause the XML file to be saved in the built assembly.



5) Add the following code when you need to load the palette definition.
You might choose the load the resource at the time your form is loaded, or maybe wait until the user chooses any appropriate action. Whatever the case may be you only need the following three lines of code to load the palette XML file from resources into a KryptonPalette instance.

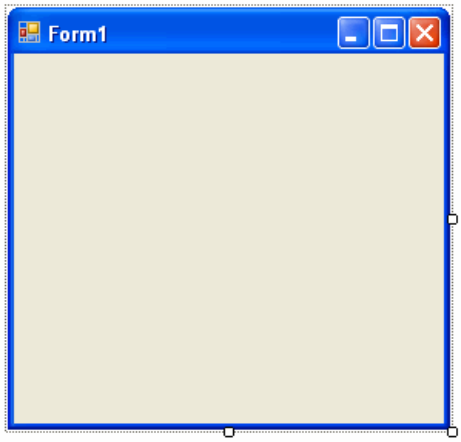
```
// Get the assembly that this type is inside
Assembly a = Assembly.GetAssembly(typeof(Form1));

// Load the named resource as a stream
Stream s = a.GetManifestResourceStream("WindowsApplication1.Blogger.xml");

// Import the stream into the KryptonPalette instance
kryptonPalette1.Import(s);
```

Of course, you will need to alter the 'typeof(Form1)' to be a type that is defined in the assembly that contains the embedded resource. Then you will also need to change the 'WindowsApplication1.Blogger.xml' string so that it has the correct path to the embedded resource.
In our simple example the palette XML file was added as a top level item of the project. Therefore the string is constructed by placing the assembly namespace followed with the name of the file. If your assembly has a namespace of 'MyCompany.MyProject' then the string would have been 'MyCompany.MyProject.Blogger.xml'. You can discover the namespace used by looking at the project properties.

Note that if the resource is added inside a folder then you need to include the folder name in the created resource path. So if your project has a 'Resources' folder that you place the palette inside then your path would become 'MyCompany.MyProject.Resources.Blogger.xml'. If you have trouble getting the path correct then use the 'ildasm' command line utility on your built assembly to discover the stored path inside the assembly.

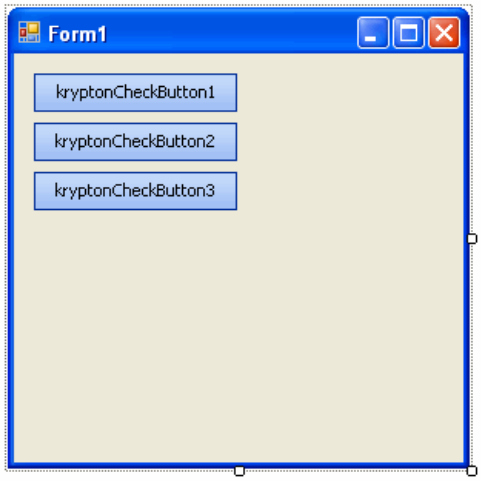
Tutorial – Multiple Choice Buttons

1) Create a new Windows Forms project
This will automatically create a form in design mode as below.

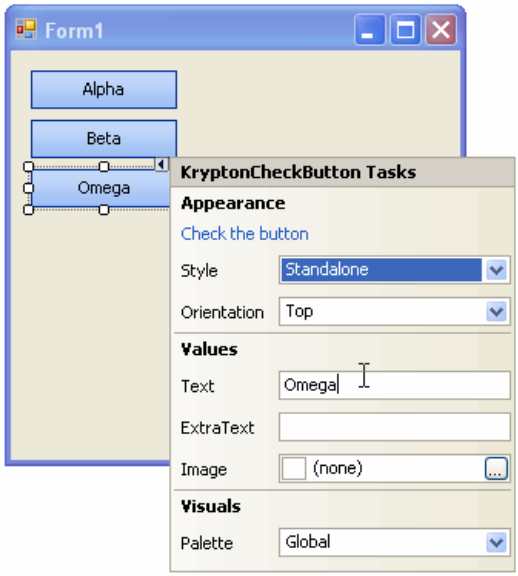


2) Ensure that the Krypton Toolkit components are in the Toolbox
If not the Using Krypton in VS2005 tutorial can be used to add them.

3) Drag and drop three KryptonCheckButton instances onto the form
Place the controls in a vertical line as shown below.

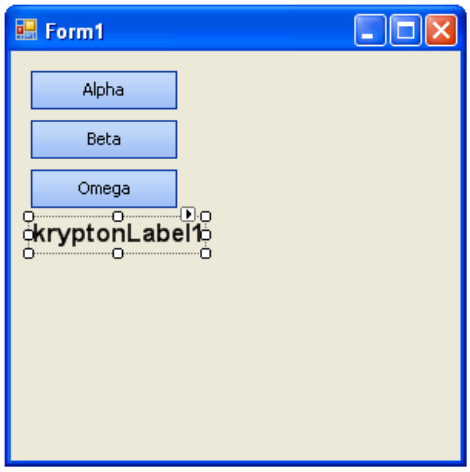


4) Use the KryptonCheckButton smart tags to enter 'Text' values of 'Alpha', 'Beta' and 'Omega'
You can use the properties window to change the 'Text' values but the smart tag provides easier access to the common values and operations on all Krypton controls. As you can see below you just need to click the small arrow on the top right of each control in order to display the smart tag for the control.



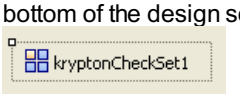
5) Drag and drop a KryptonLabel instance below the check buttons

We will use the label control for feedback on the current choice selected.



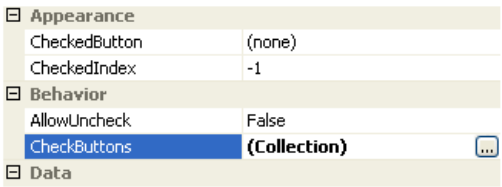
6) Drag and drop a KryptonCheckSet instance onto the form

The KryptonCheckSet set is used to manage the exclusive selection logic and is a component placed onto the component tray area. When you drop the component onto the form it will not create a control but instead a component as shown below in the area at the bottom of the design screen.



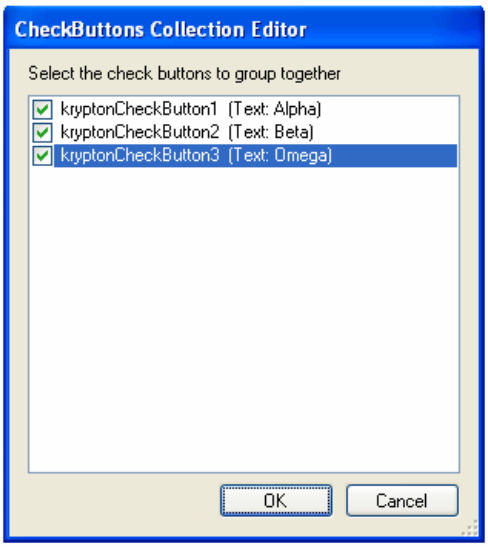
7) Find the 'CheckButtons' property in the properties window and press the '…' button

The '...' button as shown below indicates that the collection will be edited by showing a modal dialog.



8) Select all three check button entries and press 'OK'

The collection editor for the KryptonCheckSet component allows the user to specify which of the KryptonCheckButton instances it should manage. As you can see below, all three of the controls we added earlier are displayed along with the current text they are showing. You should click the check box for each entry so they are all ticked.



9) Double click the 'kryptonCheckSet1' component

This will generate a default event handler for the CheckedButtonChanged event that we are using to update the label control with details of the new selection. Enter the following code into the handler.

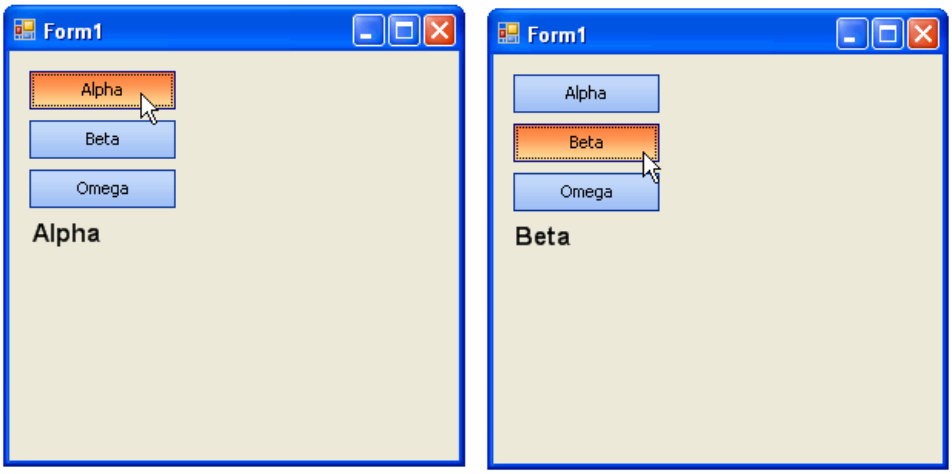
```
private void kryptonCheckSet1_CheckedButtonChanged(object sender,
                                                   EventArgs e)
{
      KryptonCheckButton cb = (KryptonCheckButton)kryptonCheckSet1.CheckedButton;
      kryptonLabel1.Text = cb.Text;
}
```

Then add the following statement to the top of the source file so that it will compile.

```
using ComponentFactory.Krypton.Toolkit;
```

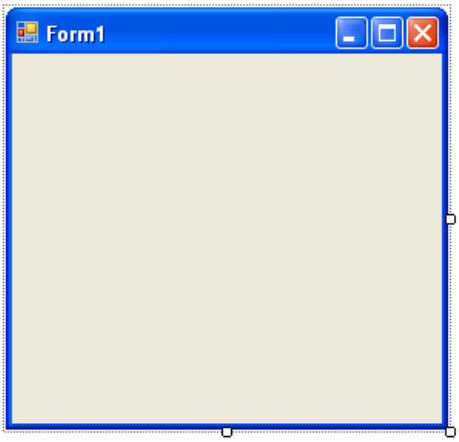
10) Compile and run the application

As you click the different check buttons the check set is automatically ensuring that only one button at a time is checked and then generating the checked changed event so that we can update the label with the new selection. You can see below the results of clicking the first and second check buttons.



Tutorial – 3 Pane Application

1) Create a new Windows Forms project
This will automatically create a form in design mode as below.



2) Add a reference to the ComponentFactory.Krypton.Toolkit assembly
C# : Right click the 'References' group in your project and select the 'Add Reference...' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.
VB.NET : Right click the project in the 'Solution Explorer' window and choose the 'Add Reference' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.

3) Ensure that the Krypton Toolkit components are in the Toolbox
If not the Using Krypton in VS2005 tutorial can be used to add them.

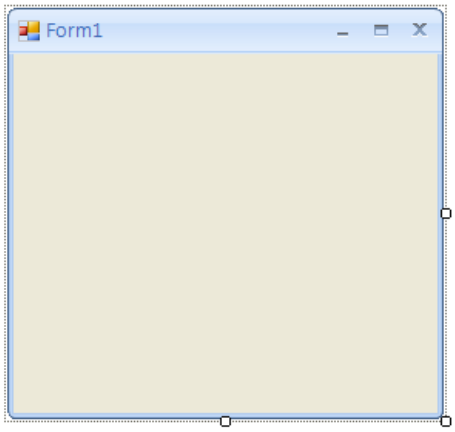
4) Open the code view for the form and change the base class.
Change the base class from the default of 'Form' to be 'ComponentFactory.Krypton.Toolkit.KryptonForm'. Your new definition for C# would be: -

```
public partial class Form1 : ComponentFactory.Krypton.Toolkit.KryptonForm
```

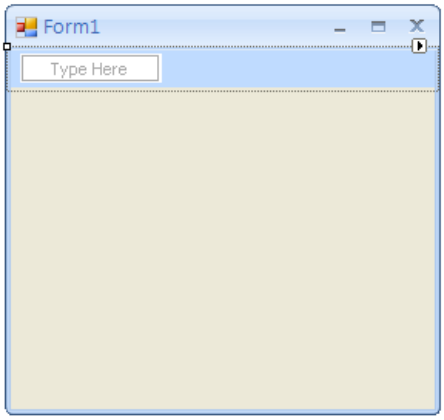
If using VB.NET then your new definition should like this: -

```
Partial Class Form1
    Inherits ComponentFactory.Krypton.Toolkit.KryptonForm
```

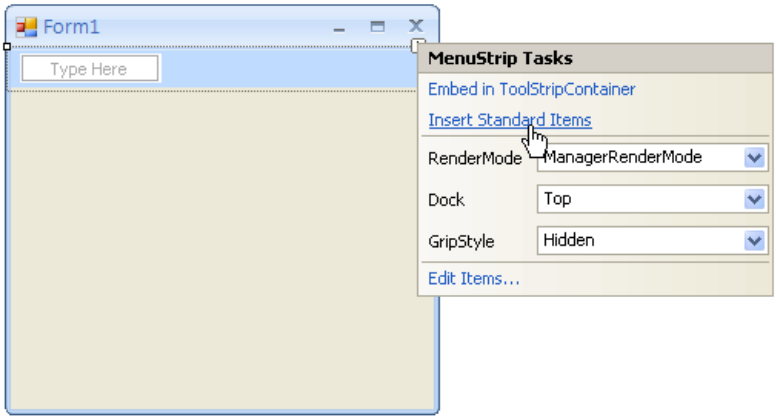
Recompile the project and then show the form in design mode again, this time you should see custom chrome applied to the form.



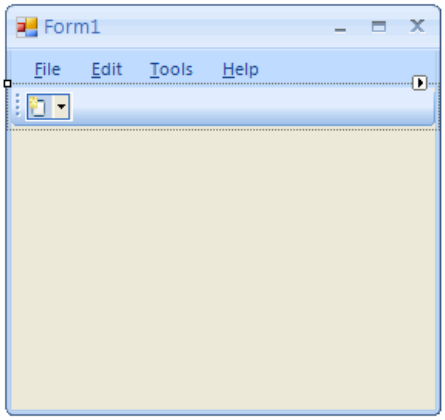
5) Drag a MenuStrip from the toolbox and drop it on the form
This will automatically dock itself to the top of the form.



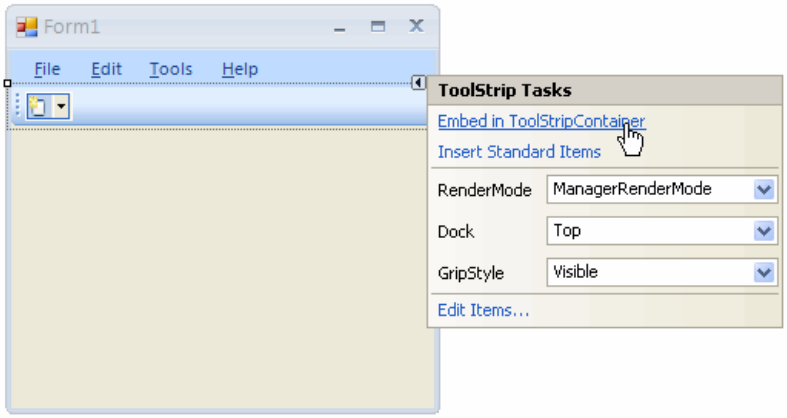
6) Use the MenuStrip smart tag and click 'Insert Standard Items'
Click the small box on the top right of the MenuStrip to open the smart tag.



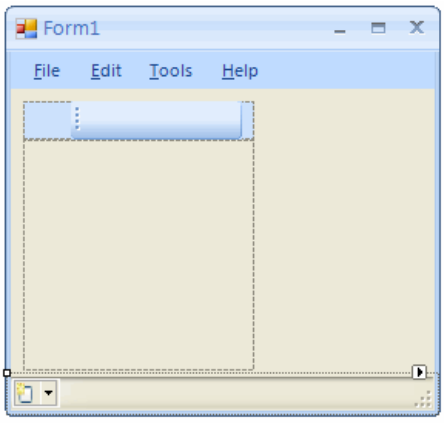
7) Drag a ToolStrip from the toolbox and drop it on the form
This will automatically dock itself underneath the MenuStrip



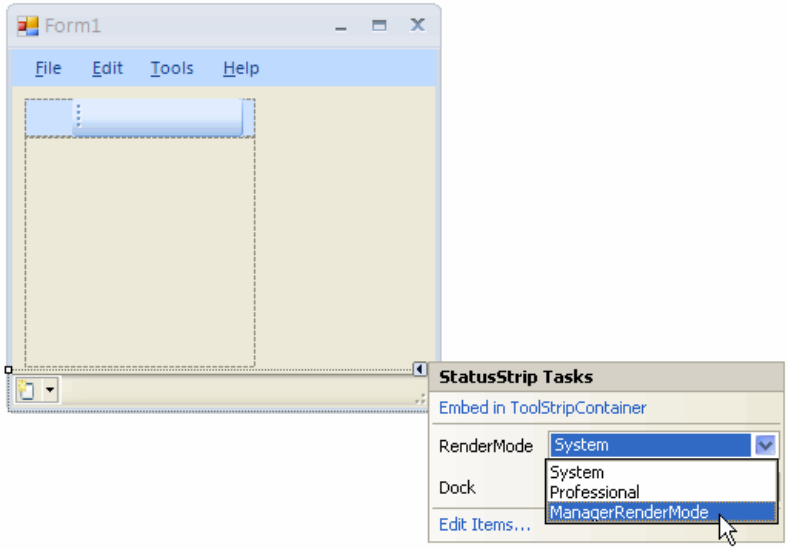
8) Use the ToolStrip smart tag and click 'Embed in ToolStripContainer'
This will create a box with four edge markers and a tool bar area at the top.



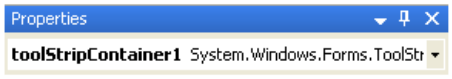
9) Drag a StatusStrip from the toolbox and drop it on the form
This will automatically be docked to the bottom of the form as shown here.



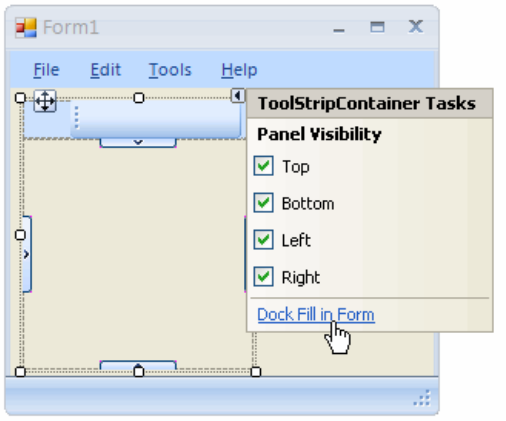
10) Use the StatusStrip smart tag and change the 'RenderMode' to 'ManagerRenderMode'
By default the StatusStrip uses the system renderer but Krypton needs to have all tool strips use the global ToolStripManager renderer in order to ensure a consistent look and feel across all the strips. So we change the rendering mode to ensure the consistent appearance.



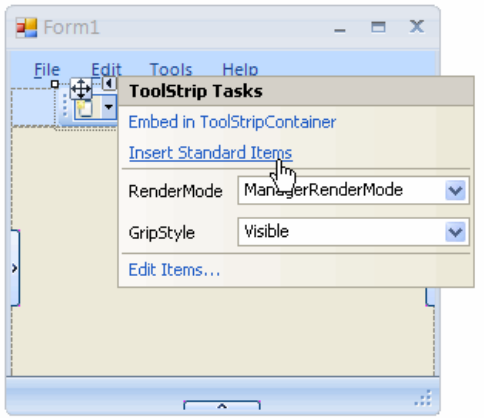
11) Use the properties window to select the 'toolStripContainer1'
You cannot select the tool strip container by clicking on the container itself in the designer and so we need to use the properties window to manually cause the selection to change. Once selected you will notice the designer view change to show the entire tool strip container selected with the smart tag button displayed.



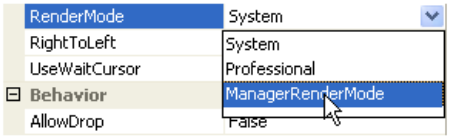
12) Use the ToolStripContainer smart tag and select 'Dock Fill in Form'
We do this so the container takes up all the space left over after positioning the menu and status strips.



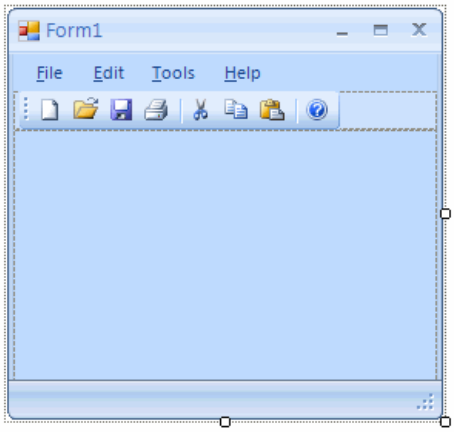
13) Select the tool strip and use the smart tag to select 'Insert Standard Items'
Click the small box on the top right of the ToolStrip to open the smart tag.



14) Click center of the client area then use properties window to set 'RenderMode' to 'ManagerRenderMode'
By default the content panel in the center of the tool strip container uses the system renderer but Krypton needs to have all tool strips use the global ToolStripManager renderer in order to ensure a consistent look and feel across all areas. So we change the rendering mode to ensure the consistent appearance.



You should now have the following appearance which is the starting point for building the specific functionality of this tutorial.

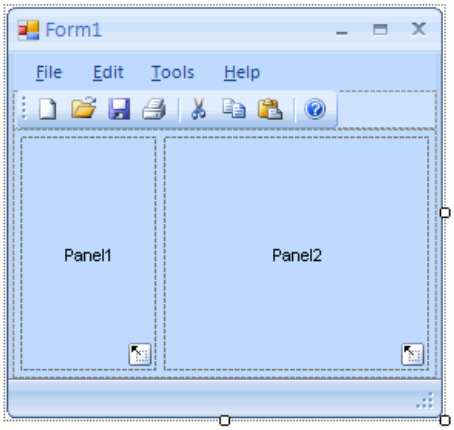


15) Modify the Padding property for the 'toolStripContainer1.ContentPanel' to 5 on all sides
You should still have the content panel of the tool strip container selected in the properties window, if not then just click in the center of the client area and the content panel will be selected again. Then alter the padding property as shown here. This adds padding around

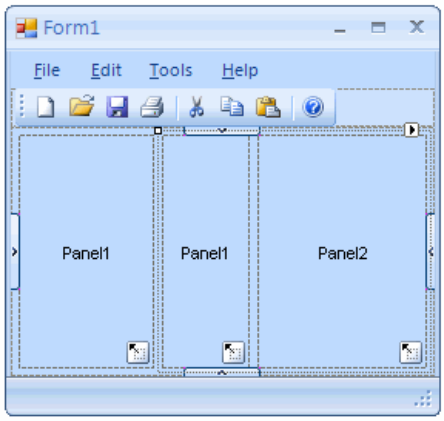
the four form edges otherwise all the panels would be placed hard against the edges.

Padding	5, 5, 5, 5
All	5
Left	5
Top	5
Right	5
Bottom	5

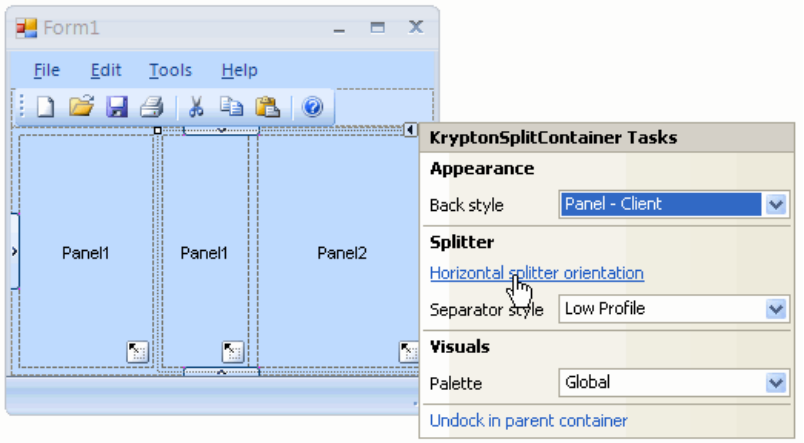
16) Drag a KryptonSplitContainer from the toolbox and drop it inside the KryptonPanel
You should see text indicating the position of Panel1 and Panel2 areas.



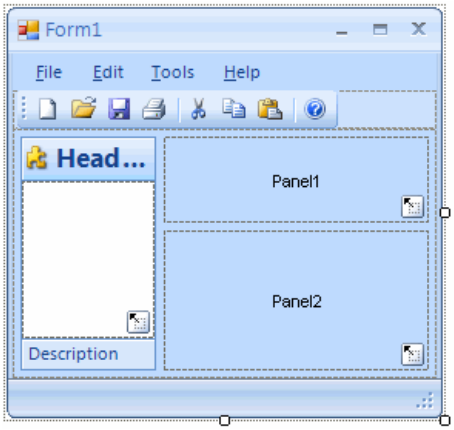
17) Drag a KryptonSplitContainer from the toolbox and drop it inside the area marked 'Pane2'
There will now be two areas marked as Panel1 but only one marked as Panel2.



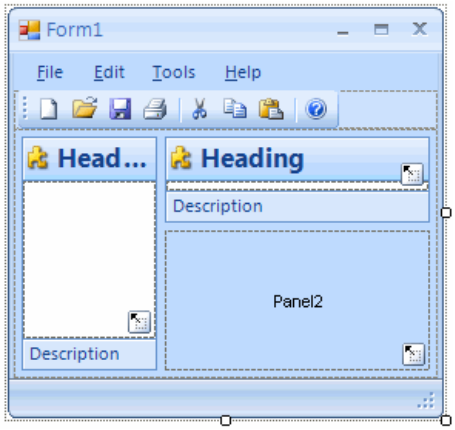
18) Use the smart tag for the second KryptonSplitContainer and click 'Horizontal splitter orientation'
This will switch the splitter from being vertical to horizontal in direction.



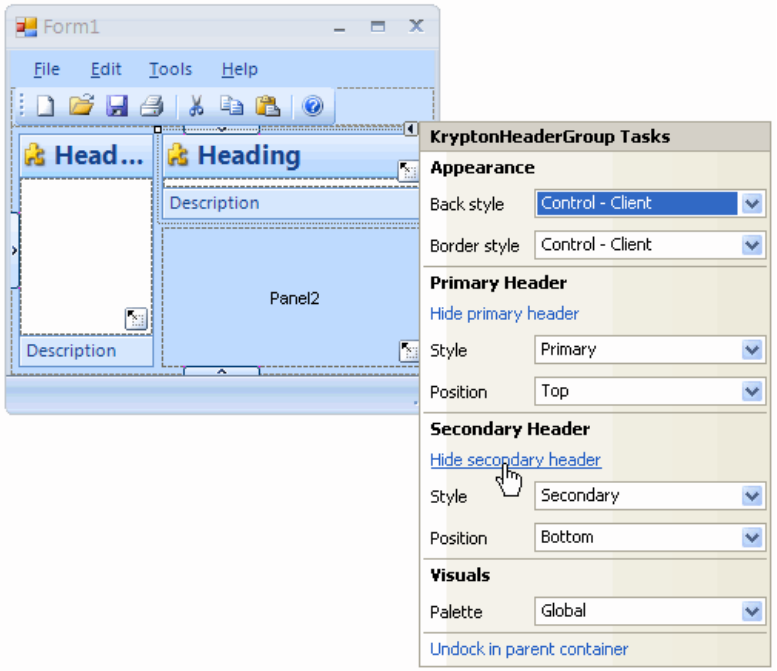
19) Drag a KryptonHeaderGroup from the toolbox and drop onto the left most 'Panel1'
The first pane needs a top and bottom header so we use a KryptonHeaderGroup. Once dropped you will need to use the smart tag of the header group in order to select the 'Dock in parent container' so that it fills the entire 'Panel1' area it was dropped into.



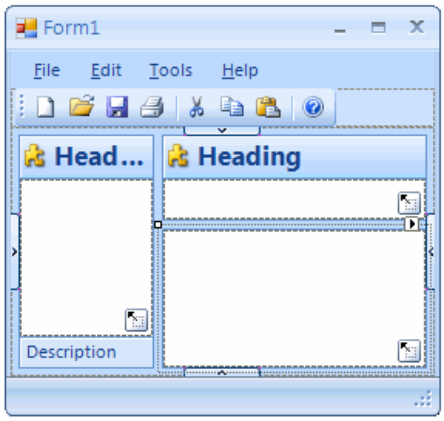
20) Drag a KryptonHeaderGroup from the toolbox and drop onto the top most 'Panel1'
The second pane needs one header so we still need to use a KryptonHeaderGroup. Once dropped you will need to use the smart tag of the header group in order to select the 'Dock in parent container' so that it fills the entire 'Panel1' area it was dropped into.



21) Use the smart tag for the second KryptonHeaderGroup and click 'Hide secondary header'
We remove the bottom header from being displayed.



22) Drag a KryptonGroup from the toolbox and drop onto the 'Panel2' area
The third pane does not need any headers so a KryptonGroup is used. Once dropped you will need to use the smart tag of the group in order to select the 'Dock in parent container' so that it fills the entire 'Panel2' area it was dropped into.



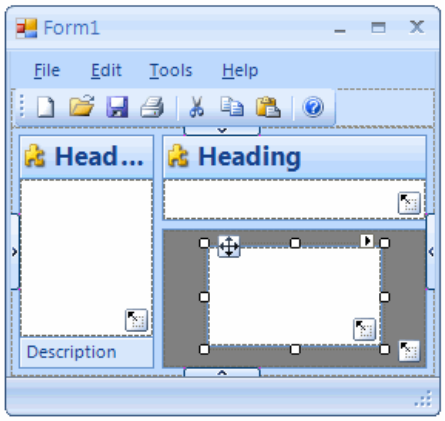
23) Modify the 'StateCommon -> Back -> Color1' property to 'AppWorkspace' We want the padding area to be distinctive and so change the color.

StateCommon	**Modified**
Back	**Modified**
Color1	**AppWorkspace**
Color2	
ColorAlign	Inherit
ColorAngle	-1
ColorStyle	Inherit
Draw	Inherit
GraphicsHint	Inherit
Image	(none)
ImageAlign	Inherit
ImageStyle	Inherit
Border	

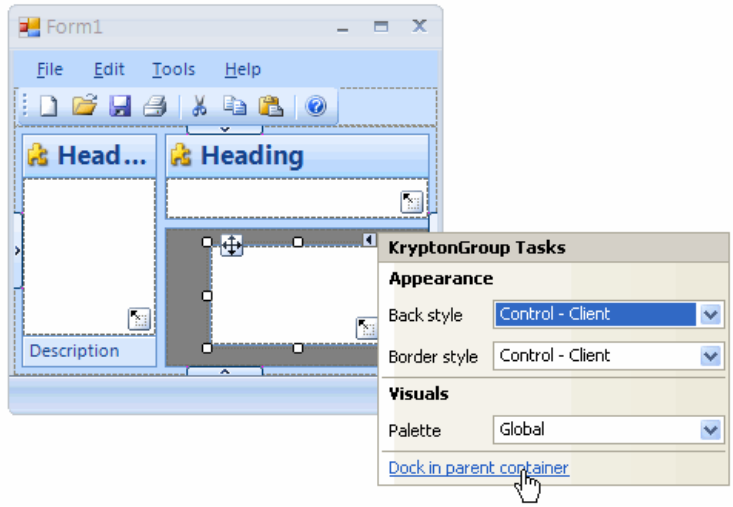
24) Modify the Padding property for the new kryptonGroup1.Panel to 5 on all sides
Click on the center of the new group to select the kryptonGroup1.Panel control. This is the Panel instance that is positioned inside the border of the group. Once the panel is selected set the border to 5 by using the properties window as can be seen below.

Padding	**5, 5, 5, 5**
All	**5**
Left	**5**
Top	**5**
Right	**5**
Bottom	**5**

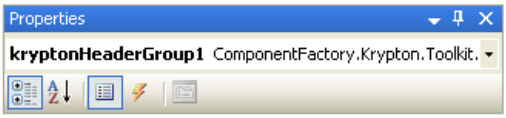
25) Drag a KryptonGroup from the toolbox and drop into the existing KryptonGroup



26) Use the smart tag for the KryptonGroup and click 'Dock in parent container'
We need the content panel to always fill the entire available area.

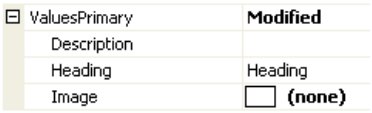


27) Select the 'kryptonHeaderGroup1' instance in the properties window

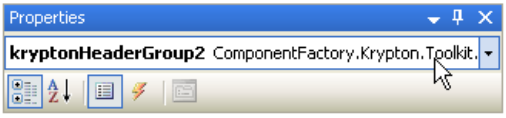


27) Set the 'ValuesPrimary -> Image' property to (none)

We do not want the left pane to have an image in our example.

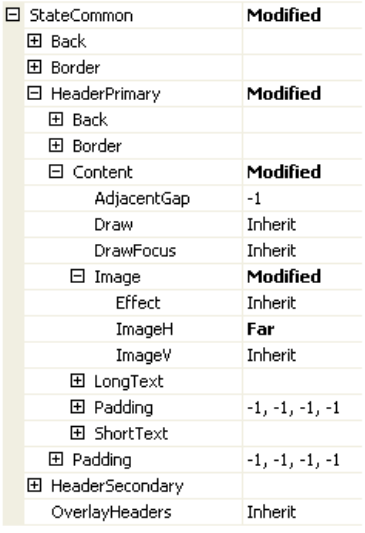


29) Select the 'kryptonHeaderGroup2' instance in the properties window

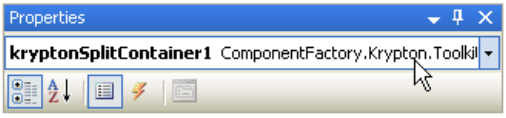


30) Modify the 'StateCommon -> HeaderPrimary -> Content -> Image –> ImageH' property to be 'Far'

By default the image appears on the left, we want to reverse this to the opposite side.

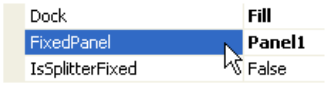


31) Select the 'kryptonSplitContainer1' instance in the properties window



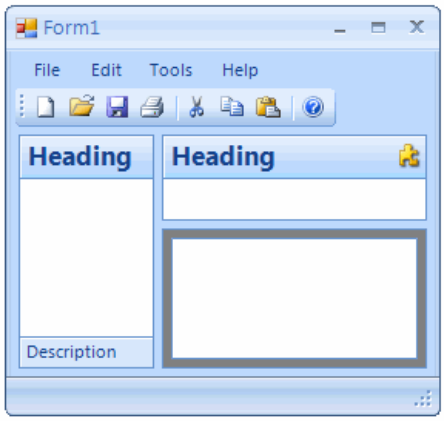
32) Modify the 'FixedPanel' property to be 'Panel1'

Now resizing the main form will not change the size of left pane.



33) Compile and run the application

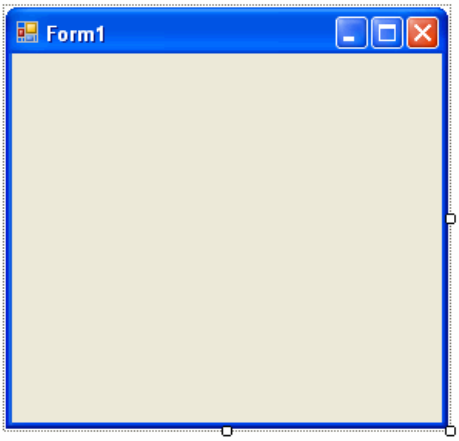
At runtime you should be able to resize the main form and notice that the left pane stays a constant width as the right hand panes are modified. The top and bottom panes on the right hand side will retain the same relative sizing as the height of the form is changed.



Tutorial – Expanding HeaderGroups (Splitters)

Part 1 - Expanding/Collapsing Left Panel

1) Create a new Windows Forms project
This will automatically create a form in design mode as below.



2) Add a reference to the ComponentFactory.Krypton.Toolkit assembly
C# : Right click the 'References' group in your project and select the 'Add Reference...' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.
VB.NET : Right click the project in the 'Solution Explorer' window and choose the 'Add Reference' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.

3) Ensure that the Krypton Toolkit components are in the Toolbox
If not the Using Krypton in VS2005 tutorial can be used to add them.

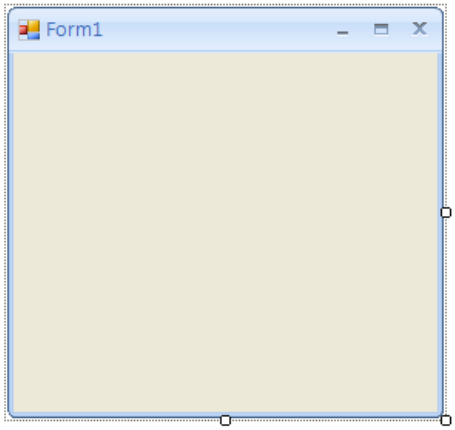
4) Open the code view for the form and change the base class.
Change the base class from the default of 'Form' to be 'ComponentFactory.Krypton.Toolkit.KryptonForm'. Your new definition for C# would be: -

```
public partial class Form1 : ComponentFactory.Krypton.Toolkit.KryptonForm
```

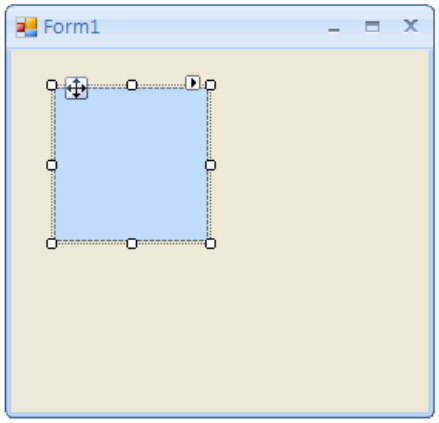
If using VB.NET then your new definition should like this: -

```
Partial Class Form1
    Inherits ComponentFactory.Krypton.Toolkit.KryptonForm
```

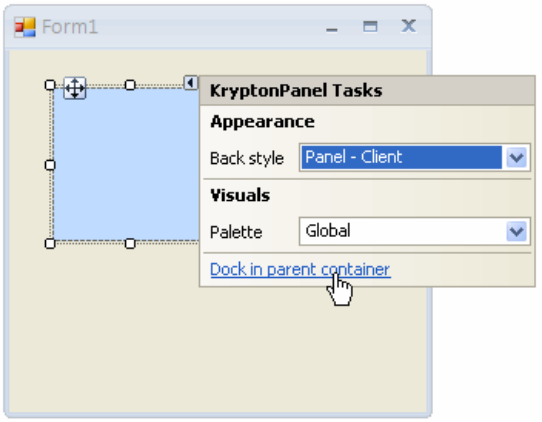
Recompile the project and then show the form in design mode again, this time you should see custom chrome applied to the form.



5) Drag a KryptonPanel from the toolbox and drop it in the centre of the form
When dropped it should look like the following picture.



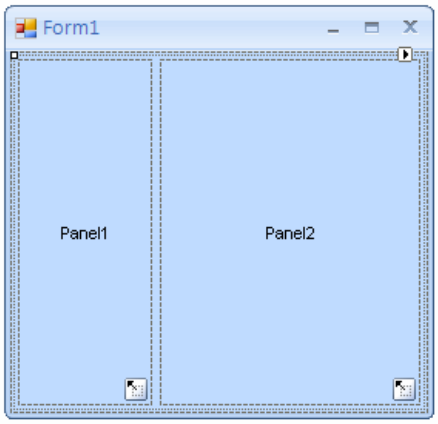
6) Use the KryptonPanel smart tag and click 'Dock in parent container'
The panel will now occupy the entire client area even when the form is resized.



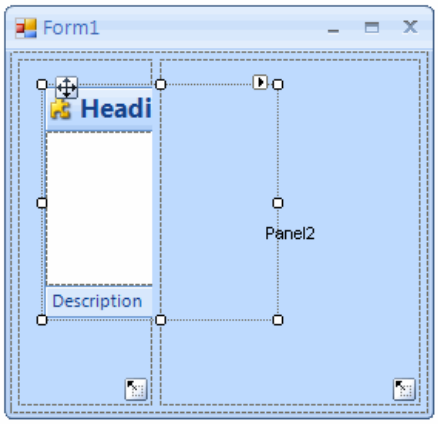
7) Modify the Padding property for the new KryptonPanel to 5 on all sides
We need to add padding because another control will be placed inside and we want a nice border around the contained control.

Padding	5, 5, 5, 5
All	5
Left	5
Top	5
Right	5
Bottom	5

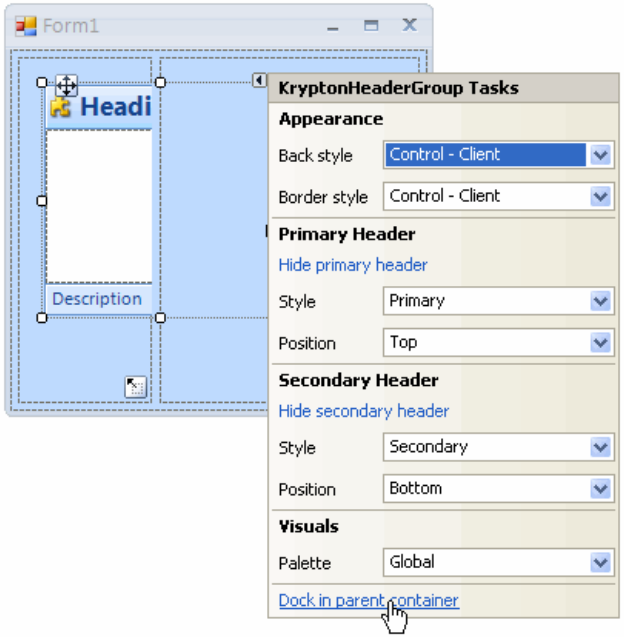
8) Drag a KryptonSplitContainer from the toolbox and drop it inside the KryptonPanel
You should see text indicating the position of Panel1 and Panel2 areas.



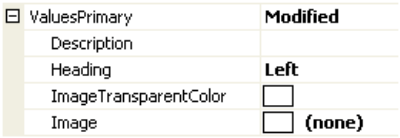
9) Drag a KryptonHeaderGroup from the toolbox and drop it inside Panel1



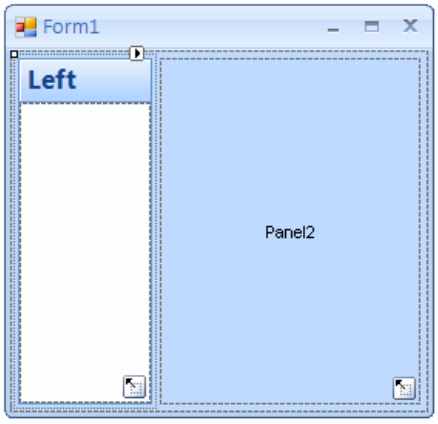
10) Use the KryptonHeaderGroup smart tag and click 'Dock in parent container'
The panel will now occupy the entire client area of Panel1.



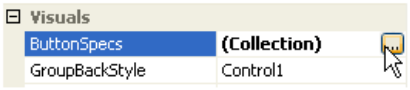
11) Modify the KryptonHeaderGroup 'Text' and 'Image' properties
Use the properties window to find the 'ValuesPrimary' property and then set the 'Text' to 'Left' and remove the 'Image' value.



12) Modify the KryptonHeaderGroup 'HeaderVisibleSecondary' property to 'False'
This will remove from display the second header and leave the display as follows.

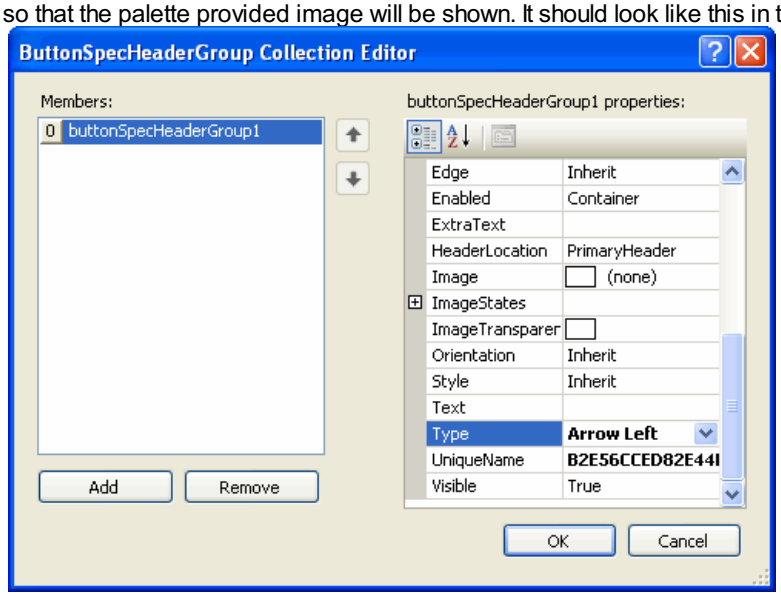


13) Find the 'ButtonSpecs' property and click the ellipses button
We we will use the collection property to define the header buttons.



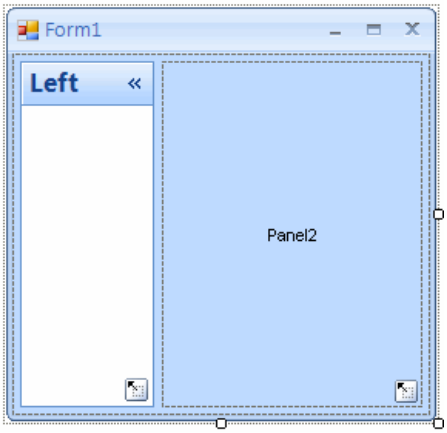
14) Add one instance to the collection and set the 'Type' to 'Arrow Left'

Use the 'Add' button on the collection editor to add a new *ButtonSpec* entry. Then set the 'Type' property of the entry to be 'Arrow Left' so that the palette provided image will be shown. It should look like this in the dialog box, then press the 'OK' button.



15) Initial setup of Left group is completed

You should now see the following picture as the initial display for the left group.



16) Enter code view and add the following using statement

This is needed to ensure that the rest of the code will compile.

```
using ComponentFactory.Krypton.Toolkit;
```

17) Hook into the header button click event

Modify the constructor for the form with the following code in order to be notified when the user clicks the header button.

```
public Form1() {
   InitializeComponent();

   // Hook into the click events on the header buttons
   kryptonHeaderGroup1.ButtonSpecs[0].Click += new EventHandler(OnLeftRight);
}
```

18) Add private field for remembering header width

Add the following private field to the form class.

```
private int _widthLeftRight;
```

19) Add button click handler code

Add the following event handler to the form class.
A description of how the code works follows after the code.

```
private void OnLeftRight(object sender, EventArgs e) {
```

```
    // (1) Suspend changes until all splitter properties have been updated
    kryptonSplitContainer1.SuspendLayout();

    // (2) Is the left header group currently expanded?
    if (kryptonSplitContainer1.FixedPanel == FixedPanel.None)    {
        // (3) Make the left panel fixed in size
        kryptonSplitContainer1.FixedPanel = FixedPanel.Panel1;
        kryptonSplitContainer1.IsSplitterFixed = true;

        // (4) Remember the current height of the header group
        _widthLeftRight = kryptonHeaderGroup1.Width;

        // (5) Find the new width to use for the header group
        int newWidth = kryptonHeaderGroup1.PreferredSize.Height;

        // (6) Make the header group fixed just as the new height
        kryptonSplitContainer1.Panel1MinSize = newWidth;
        kryptonSplitContainer1.SplitterDistance = newWidth;

        // (7) Change header to be vertical and button to near edge
        kryptonHeaderGroup1.HeaderPositionPrimary = VisualOrientation.Right;
        kryptonHeaderGroup1.ButtonSpecs[0].Edge = PaletteRelativeEdgeAlign.Near;
    }
    else
    {
        // Make the bottom panel not-fixed in size anymore
        kryptonSplitContainer1.FixedPanel = FixedPanel.None;
        kryptonSplitContainer1.IsSplitterFixed = false;

        // Put back the minimize size to the original
        kryptonSplitContainer1.Panel1MinSize = 100;

        // Calculate the correct splitter we want to put back
        kryptonSplitContainer1.SplitterDistance = _widthLeftRight;

        // Change header to be horizontal and button to far edge
        kryptonHeaderGroup1.HeaderPositionPrimary = VisualOrientation.Top;
        kryptonHeaderGroup1.ButtonSpecs[0].Edge = PaletteRelativeEdgeAlign.Far;
    }

    kryptonSplitContainer1.ResumeLayout();
}
```

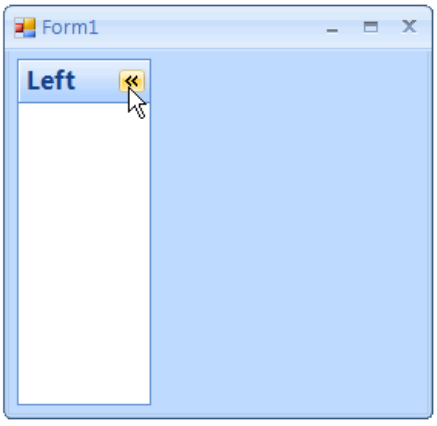
(1) Because we are changing several different splitter properties in one go we wrap the changes in a SuspendLayout() / ResumeLayout() pair so that the changes are not acted on until all the properties have been modified. This reduces the amount of flicker that would otherwise occur because each property change updates the appearance in turn.
(2) The code begins by checking if the header group is in the expanded or collapsed state. When expanded there will be no fixed panel because the user is allowed to move the separator to update the relative spacing of the two splitter panels.
(3) When becoming collapsed we want the splitter panel to become fixed in size and prevent the user from moving the separator. The first line of code will fix the first panel so that any resizing of the form will cause the second splitter panel to be resized and leave the first splitter panel the same constant size. The second line makes the separator read only.

(4) We need to remember the current width of the first splitter panel so that when the group is expanded again in the future we can put it back to the same size as it originally started at.

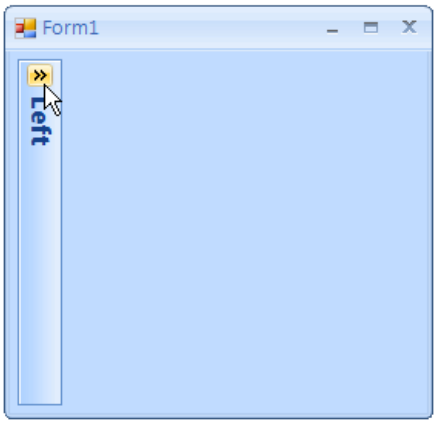
(5) Because we are going to change the header group orientation to vertical the new width of the splitter panel needs to be the height of the primary header on the header group. This height can be discovered by asking for the PreferredSize of the control now that that collapsed state has changed. The collapsed is automatically toggled because we are using an arrow button type.
(6) In order to collapse the splitter panel we need to set the new width of the first panel.

(7) Finally we change the image so that it points in the opposite direction, place it at the opposite edge of the header and change the orientation of the header to be vertical.

The 'else' section of the code is self evident once the above is understand and merely puts the header group back into the original state. At runtime the initial view of the window would be as follows.

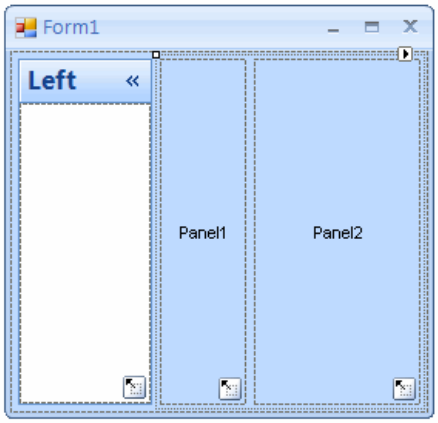


After using the header button it will then collapse down to the following.

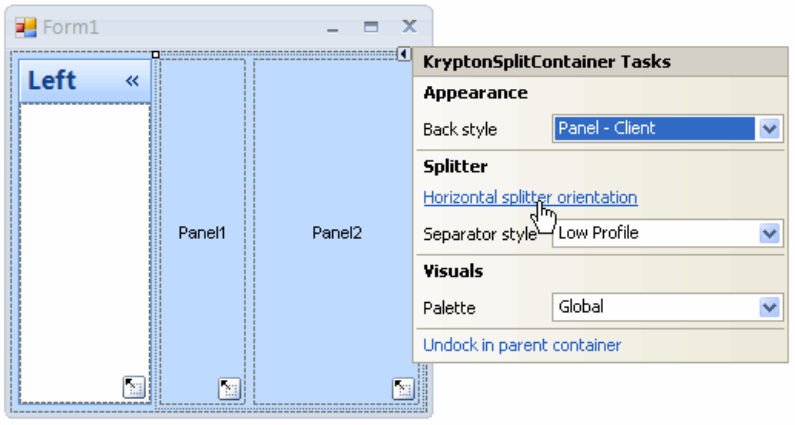


Part 2 - Expanding/Collapsing Right Panel

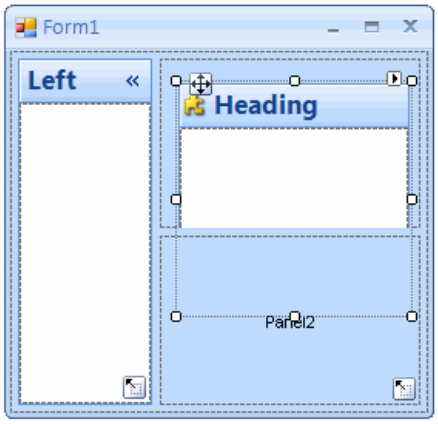
20) Drag a KryptonSplitContainer from the toolbox and drop it inside the right panel
You should see text indicating the position of new Panel1 and Panel2 areas.



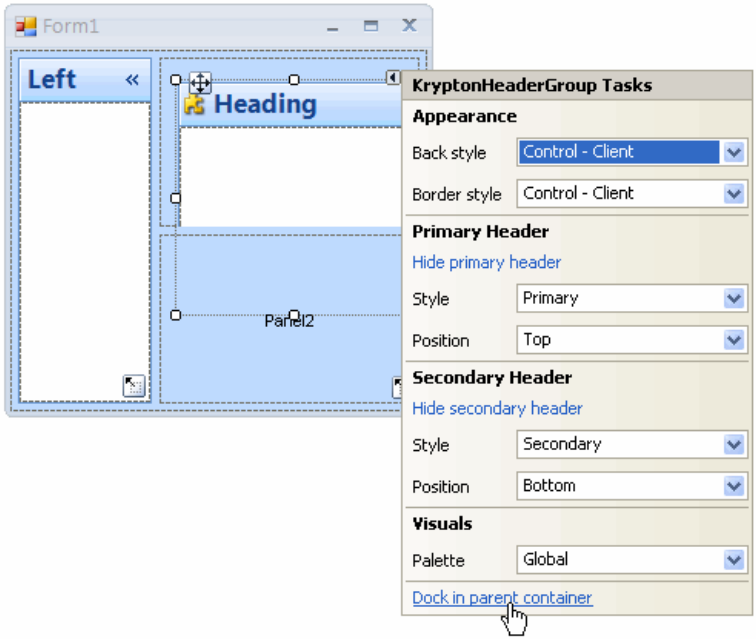
21) Use the KryptonSplitContainer smart tag and click 'Horizontal splitter orientation'
The splitter will now change to a vertical orientation with panel1 at the top and panel2 at the bottom.



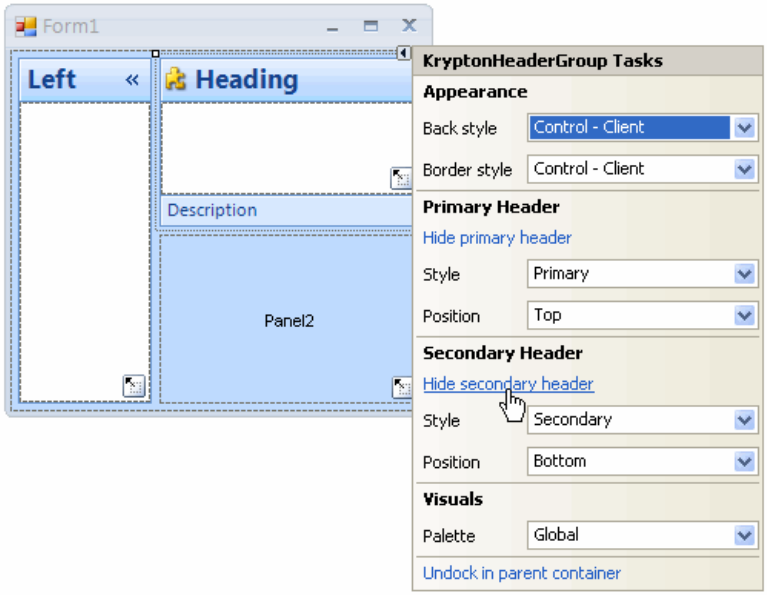
22) Drag a KryptonHeaderGroup from the toolbox and drop it inside Panel1



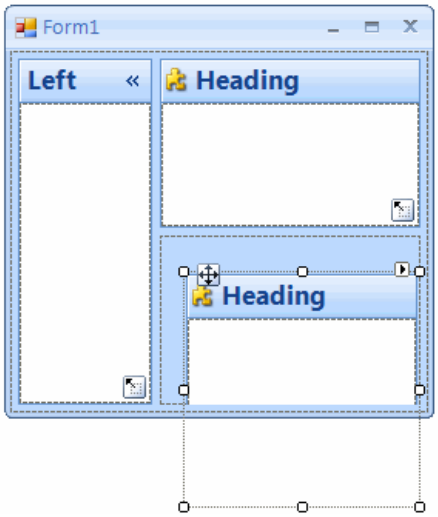
23) Use the KryptonHeaderGroup smart tag and click 'Dock in parent container'
This ensures the header group is always sized to fit the splitter panel size.



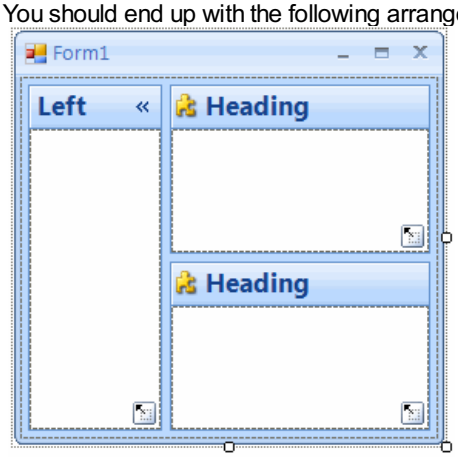
24) Use the KryptonHeaderGroup smart tag and click 'Hide secondary header'
We don't need the secondary header at the bottom for this example application.



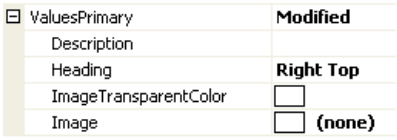
25) Drag a KryptonHeaderGroup from the toolbox and drop it inside Panel2
We now follow the same steps for the header group in the second splitter panel.



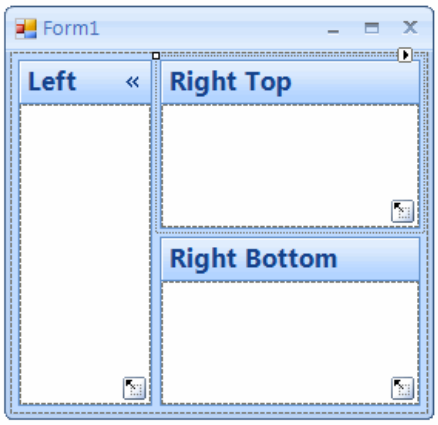
26) Use the KryptonHeaderGroup smart tag and click 'Dock in parent container'
This ensures the header group is always sized to fit the splitter panel size.
27) Use the KryptonHeaderGroup smart tag and click 'Hide secondary header'
You should end up with the following arrangement.



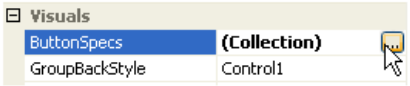
28) Modify the top right KryptonHeaderGroup 'Text' and 'Image' properties
Use the properties window to find the ValuesPrimary property and then set the 'Text' to 'Right Top' and remove the 'Image' value.



29) Modify the bottom right KryptonHeaderGroup 'Text' and 'Image' properties
Use the properties window and this time set the 'Text' to 'Right Bottom' and remove the 'Image' value.
This should result in the following updated display.

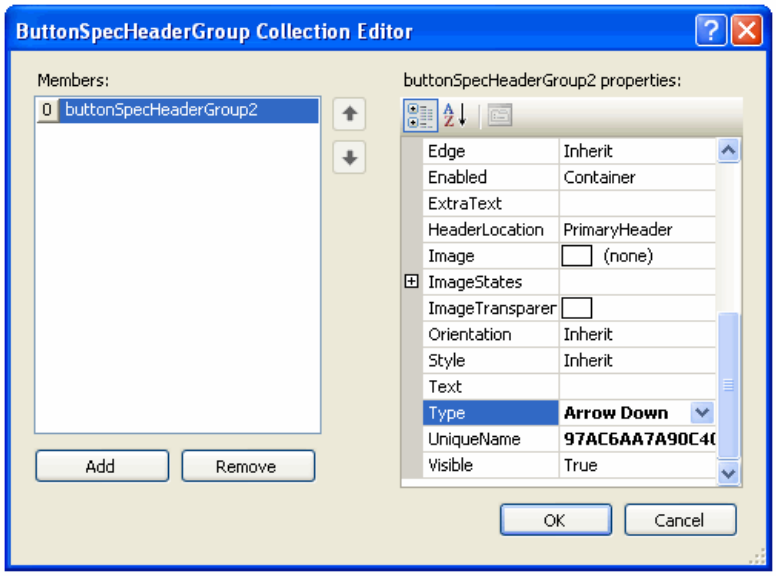


30) Find the 'ButtonSpecs' property and click the ellipses button
With the bottom right header group still selected find and click the ellipses button for the 'ButtonSpecs' property.



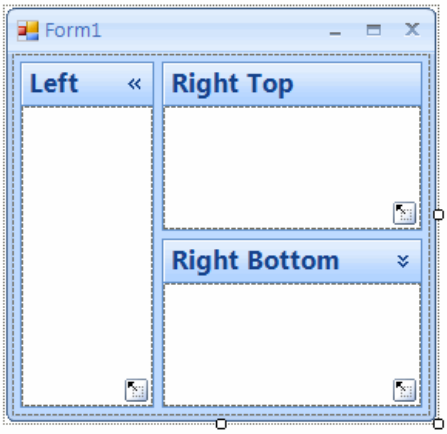
31) Add one instance to the collection and set the 'Type' to 'Arrow Down'

Use the 'Add' button on the collection editor to add a new *ButtonSpec* entry. Then set the 'Type' property of the entry to be 'Arrow Down' so that the palette provided image will be shown. It should look like this in the dialog box, then press the 'OK' button.



32) Initial setup of right groups is completed

You should now see the following picture as the initial display for the application.



33) Enter code view and hook into the button click event

Modify the constructor by adding the following line.

```
kryptonHeaderGroup3.ButtonSpecs[0].Click += new EventHandler(OnUpDown);
```

34) Add private field for remembering header height

Add the following private field to the form class.

```
private int _heightUpDown;
```

35) Add button click handler code

Add the following event handler to the form class.
A description of how the code works follows after the code.

```
private void OnUpDown(object sender, EventArgs e) {
  // (1) Is the bottom right header group currently expanded?
   if (kryptonSplitContainer2.FixedPanel == FixedPanel.None)     {
     // (2) Make the bottom panel fixed in size
      kryptonSplitContainer2.FixedPanel = FixedPanel.Panel2;
      kryptonSplitContainer2.IsSplitterFixed = true;

      // (3) Remember the current height of the header group
      _heightUpDown = kryptonHeaderGroup3.Height;

      // (4) Find the new height to use for the header group
      int newHeight = kryptonHeaderGroup3.PreferredSize.Height;

      // (5) Make the header group fixed just as the new height
```

```
     kryptonSplitContainer2.Panel2MinSize = newHeight;       kryptonSplitContainer2.SplitterDistance =
kryptonSplitContainer2.Height;
   }
   else
   {
      // Make the bottom panel not-fixed in size anymore
       kryptonSplitContainer2.FixedPanel = FixedPanel.None;
       kryptonSplitContainer2.IsSplitterFixed = false;
      // Put back the minimum size to the original
      kryptonSplitContainer2.Panel2MinSize = 100;

       // Calculate the correct splitter we want to put back
      kryptonSplitContainer2.SplitterDistance =
          kryptonSplitContainer2.Height - _heightUpDown - kryptonSplitContainer2.SplitterWidth;     }
}
```

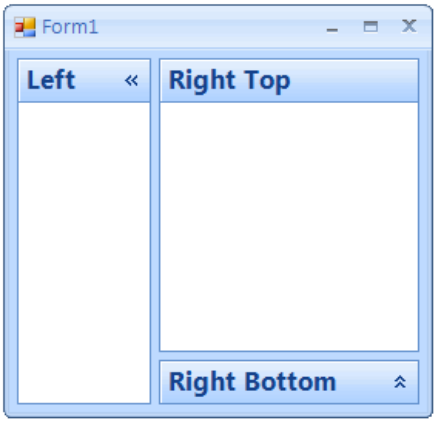
(1) The code begins by checking if the header group is in the expanded or collapsed state. When expanded there will be no fixed panel because the user is allowed to move the separator and so update the relative spacing of the two splitter panels.
(2) When becoming collapsed we want the splitter panel to become fixed in size and prevent the user from moving the separator. The first line of code will fix the second panel so that any resizing of the form will cause the first splitter panel to be resized and leave the second splitter panel the same constant size. The second line makes the separator read only.

(3) We need to remember the current height of the second splitter panel so that when the group is expanded again in the future we can put it back to the same size as it originally started at.

(4) The new height of the second splitter panel can be discovered by using the DisplayRectangle that represents the inside client area of the header group. This height can be discovered by asking for the PreferredSize of the control now that that collapsed state has changed. The collapsed is automatically toggled because we are using an arrow button type.

(5) In order to collapse the splitter panel we set the required height as the minimum height of the second split panel and then attempt to update the splitter distance to the whole height of the split container. The splitter distance will then we changed automatically by the split container control so that the minimum height of the second split panel is enforced.

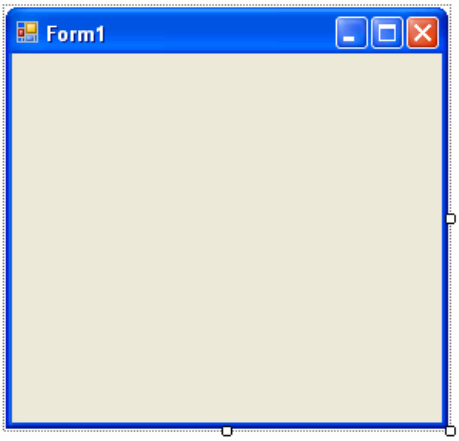
The 'else' section of the code is self evident once the above is understand and merely puts the header group back into the original state. At runtime you should be able to press the down arrow to get the following display.



© 2012 Component Factory Pty Ltd. All Rights Reserved.

Tutorial – Expanding HeaderGroups (DockStyle)

1) Create a new Windows Forms project
This will automatically create a form in design mode as below.



2) Add a reference to the ComponentFactory.Krypton.Toolkit assembly
C# : Right click the 'References' group in your project and select the 'Add Reference...' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.
VB.NET : Right click the project in the 'Solution Explorer' window and choose the 'Add Reference' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.

3) Ensure that the Krypton Toolkit components are in the Toolbox
If not the Using Krypton in VS2005 tutorial can be used to add them.

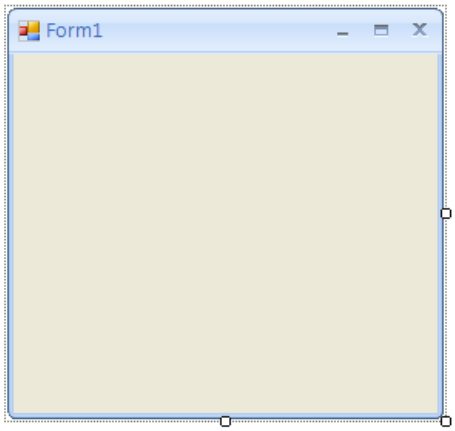
4) Open the code view for the form and change the base class.
Change the base class from the default of 'Form' to be 'ComponentFactory.Krypton.Toolkit.KryptonForm'. Your new definition for C# would be: -

```
public partial class Form1 : ComponentFactory.Krypton.Toolkit.KryptonForm
```

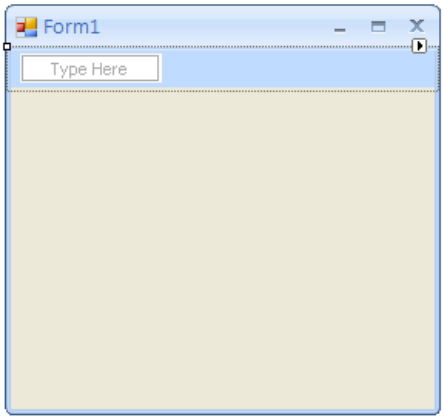
If using VB.NET then your new definition should like this: -

```
Partial Class Form1
    Inherits ComponentFactory.Krypton.Toolkit.KryptonForm
```

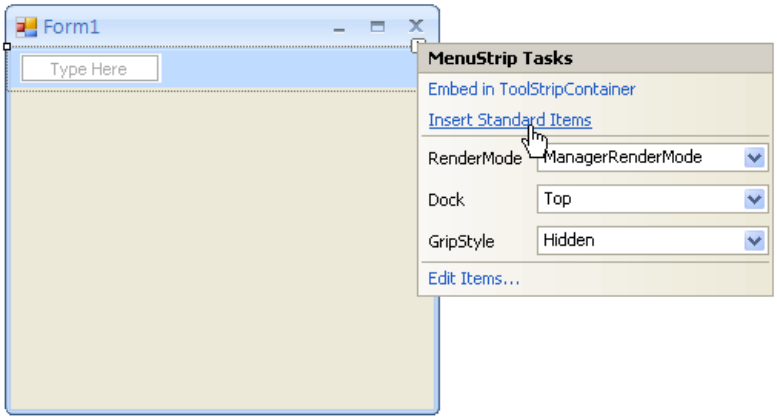
Recompile the project and then show the form in design mode again, this time you should see custom chrome applied to the form.



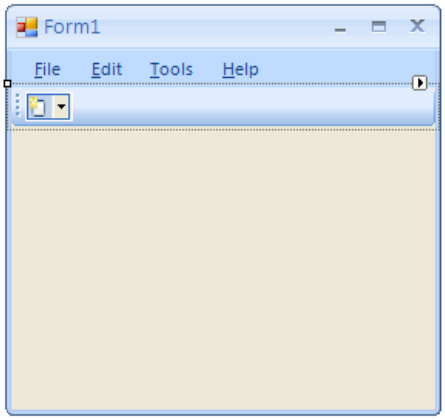
5) Drag a MenuStrip from the toolbox and drop it on the form
This will automatically dock itself to the top of the form.



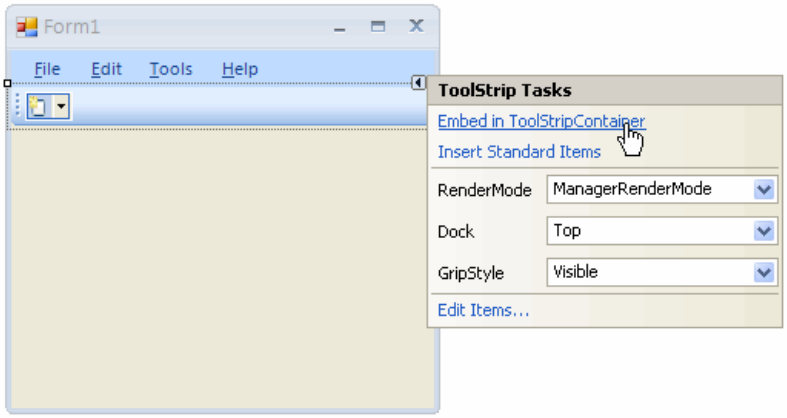
6) Use the MenuStrip smart tag and click 'Insert Standard Items'
Click the small box on the top right of the MenuStrip to open the smart tag.



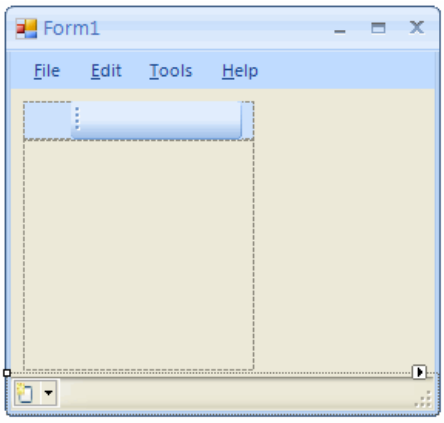
7) Drag a ToolStrip from the toolbox and drop it on the form
This will automatically dock itself underneath the MenuStrip



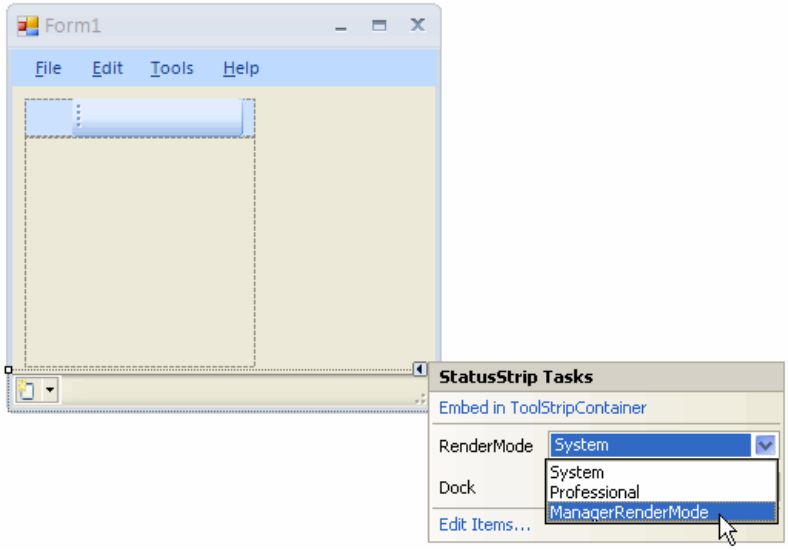
8) Use the ToolStrip smart tag and click 'Embed in ToolStripContainer'
This will create a box with four edge markers and a tool bar area at the top.



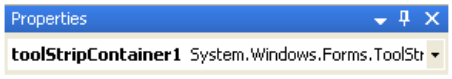
9) Drag a StatusStrip from the toolbox and drop it on the form
This will automatically be docked to the bottom of the form as shown here.



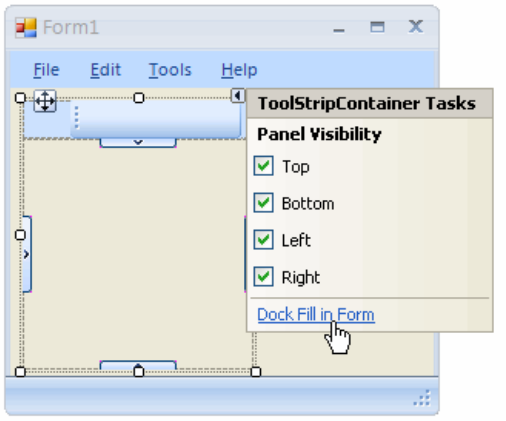
10) Use the StatusStrip smart tag and change the 'RenderMode' to 'ManagerRenderMode'
By default the StatusStrip uses the system renderer but Krypton needs to have all tool strips use the global ToolStripManager renderer in order to ensure a consistent look and feel across all the strips. So we change the rendering mode to ensure the consistent appearance.



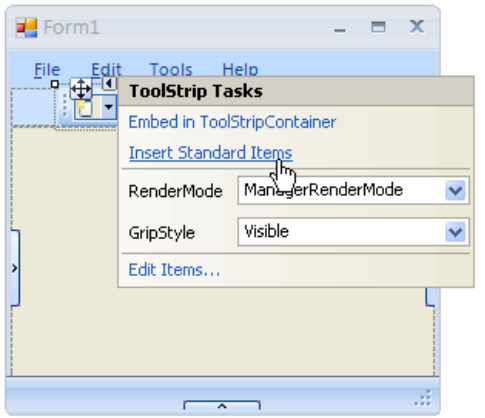
11) Use the properties window to select the 'toolStripContainer1'
You cannot select the tool strip container by clicking on the container itself in the designer and so we need to use the properties window to manually cause the selection to change. Once selected you will notice the designer view change to show the entire tool strip container selected with the smart tag button displayed.



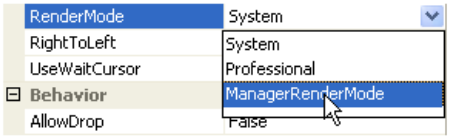
12) Use the ToolStripContainer smart tag and select 'Dock Fill in Form'
We do this so the container takes up all the space left over after positioning the menu and status strips.



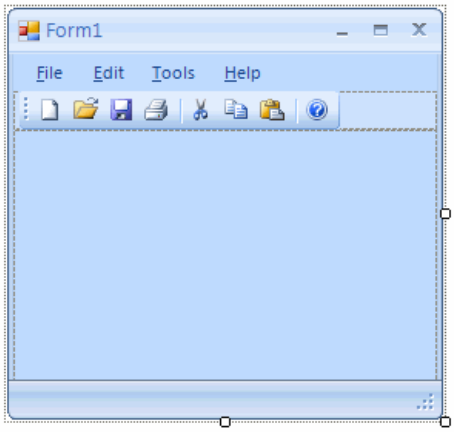
13) Select the tool strip and use the smart tag to select 'Insert Standard Items'
Click the small box on the top right of the ToolStrip to open the smart tag.



14) Click center of the client area then use properties window to set 'RenderMode' to 'ManagerRenderMode'
By default the content panel in the center of the tool strip container uses the system renderer but Krypton needs to have all tool strips use the global ToolStripManager renderer in order to ensure a consistent look and feel across all areas. So we change the rendering mode to ensure the consistent appearance.



You should now have the following appearance which is the starting point for building the specific functionality of this tutorial.

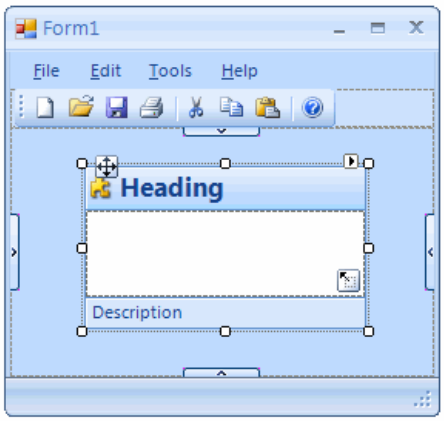


15) Modify the Padding property for the 'toolStripContainer1.ContentPanel' to 5 on all sides
You should still have the content panel of the tool strip container selected in the properties window, if not then just click in the center of the client area and the content panel will be selected again. Then alter the padding property as shown here. This adds padding around

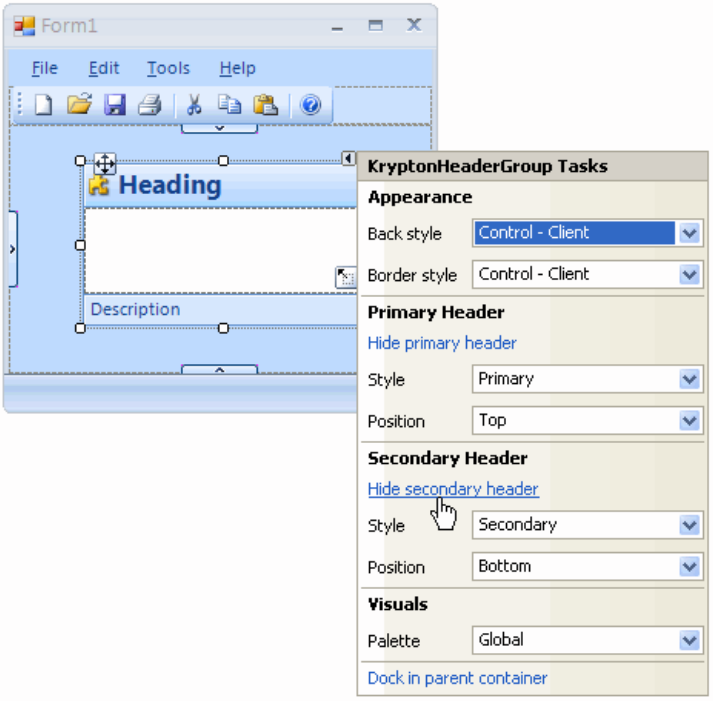
the four form edges otherwise all the panels would be placed hard against the edges.

Padding	5, 5, 5, 5
All	5
Left	5
Top	5
Right	5
Bottom	5

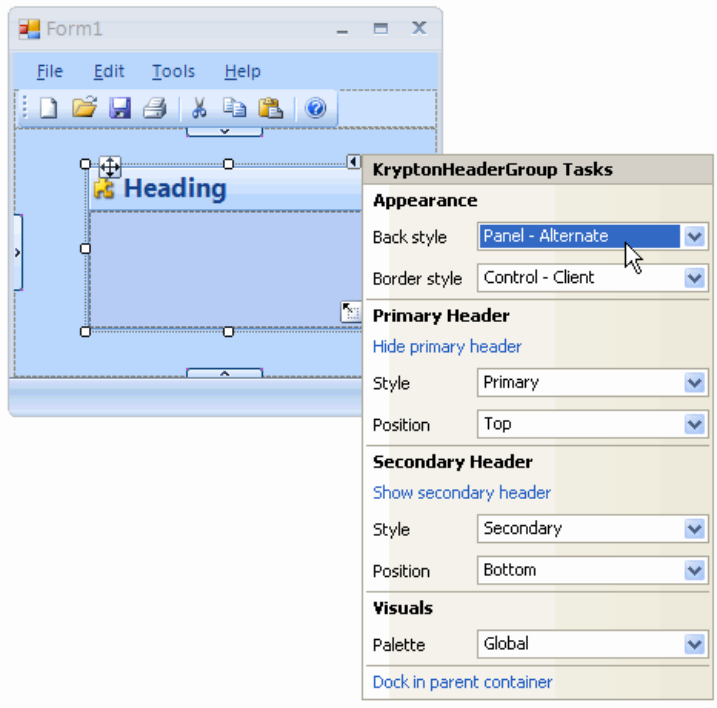
16) Drag a KryptonHeaderGroup from the toolbox and drop it onto the main panel.



17) Use the KryptonHeaderGroup smart tag and click 'Hide secondary Header'
We only need the top header for this example and so we need to hide the bottom header.

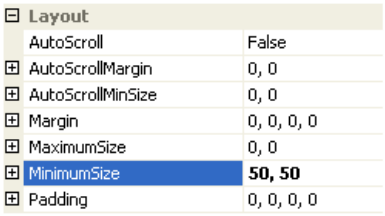


18) Use smart tag again to set the 'Back Style' to 'Panel - Alternate'
This changes to background from white to a complimentary panel style.



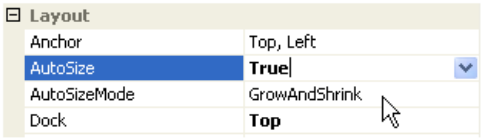
19) Select the kryptonHeaderGroup1.Panel property and set 'MinimumSize' to '50,50'

Click on the center of the header group client area to select the kryptonHeaderGroup1.Panel that is contained inside the header group. Then use the property window to alter the 'MinimumSize' to be '50,50'. This size will be enforced when the group is expanded and auto sized.

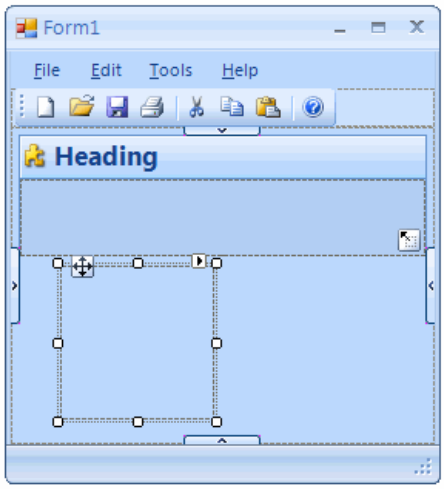


20) Select the kryptonHeaderGroup and change the 'Dock' property to 'Top' and the 'AutoSize' to 'True'

Click the header section of the header group control to show its properties again in the property window. Our example is going to place the header group at the top of the client area by using the 'Dock' property but your application could just as easily position it against any of the other edges instead. The 'AutoSize' is needed so that the header group automatically resizes appropriately when the collapsed state changes.

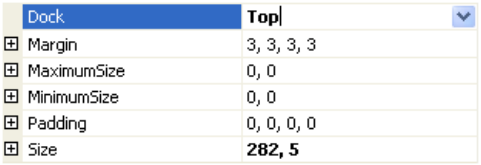


21) Drag a KryptonPanel from the toolbox and drop it onto the main panel.

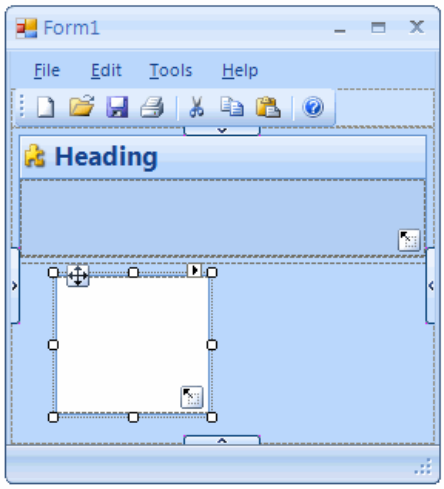


22) Set the panels 'Dock' property to be 'Top' and 'Size' to '5,5'

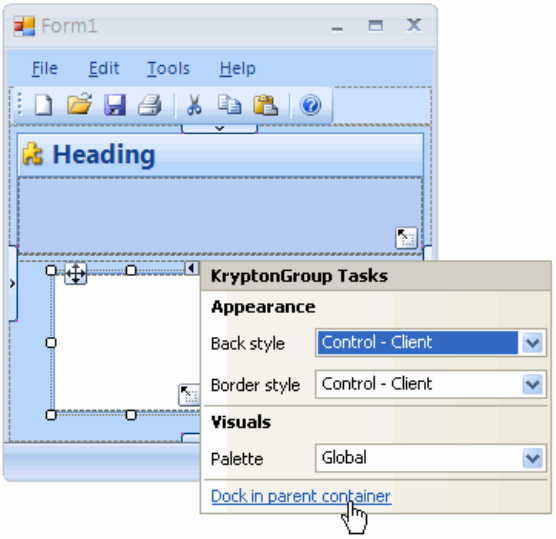
We are using the panel as spacer between the header group and the rest of the client area. So setting the 'Dock' to 'Top' places the spacer immediately below the header group and a 'Size' of '5,5' ensures the height of the spacer is 5 pixels.



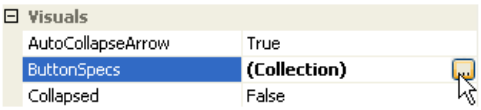
23) Drag a KryptonGroup from the toolbox and drop it onto the main panel.
This is the group we are going to use to fill the remainder of the client area.



24) Use smart tag for the KryptonGroup and click 'Dock to parent container'
We want the group to take up all the space not used by the top header group.

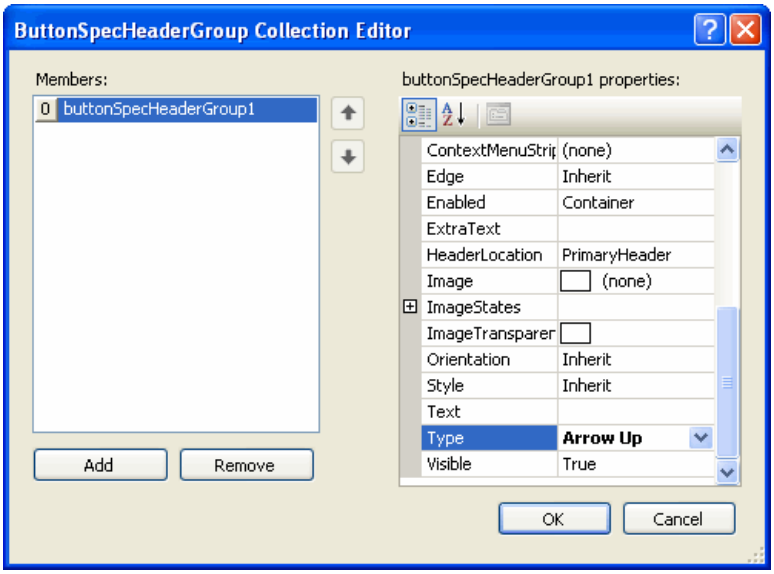


25) Select the KryptonHeaderGroup and edit the 'ButtonSpecs' collection.
Click the KryptonHeaderGroup to show the control properties in the property window. Then find the 'ButtonSpecs' property and click the edit button on the right hand side of the property, as shown below.

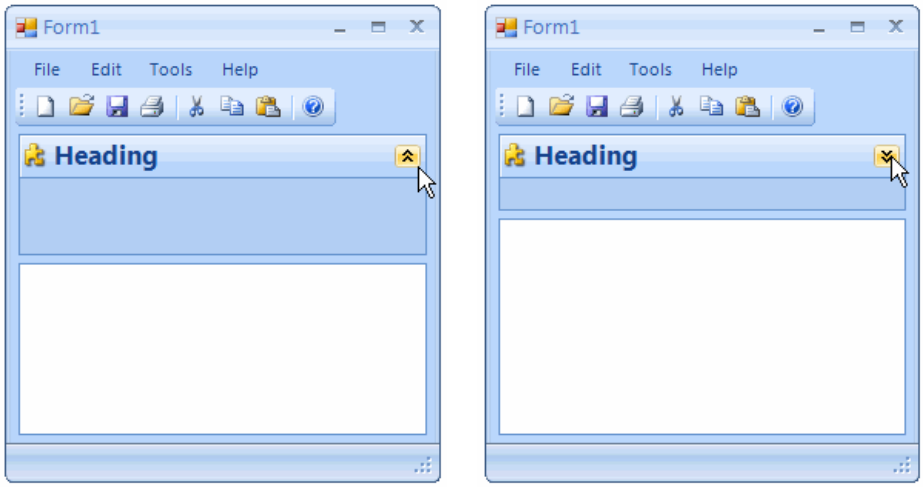


26) Use the 'Add' button on the collection editor create a new button specification.
This should create an entry in the left hand list and show a set of properties on the right.

27) Modify the 'Type' property to be 'Arrow Up'
Using this predefined type ensures the correct image is shown and pressing the button toggles the collapsed state.

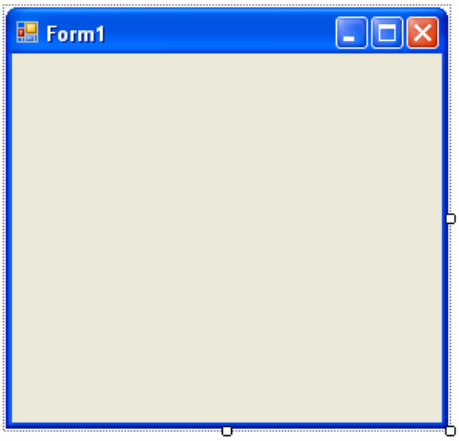


28) Compile and run the code and you will have an expanding header group.



Tutorial – Expanding HeaderGroups (Stack)

1) Create a new Windows Forms project
This will automatically create a form in design mode as below.



2) Add a reference to the ComponentFactory.Krypton.Toolkit assembly
C# : Right click the 'References' group in your project and select the 'Add Reference...' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.
VB.NET : Right click the project in the 'Solution Explorer' window and choose the 'Add Reference' option. Use the 'Browse' tab of the shown dialog box to navigate to the location you installed the library and choose the '\bin\ComponentFactory.Krypton.Toolkit.dll' file. This will then add the toolkit assembly to the list of references for the project.

3) Ensure that the Krypton Toolkit components are in the Toolbox
If not the Using Krypton in VS2005 tutorial can be used to add them.

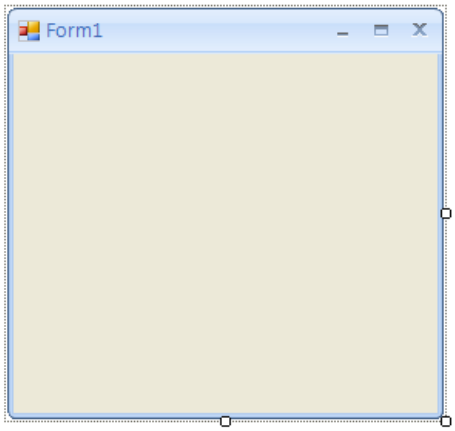
4) Open the code view for the form and change the base class.
Change the base class from the default of 'Form' to be 'ComponentFactory.Krypton.Toolkit.KryptonForm'. Your new definition for C# would be: -

```
public partial class Form1 : ComponentFactory.Krypton.Toolkit.KryptonForm
```

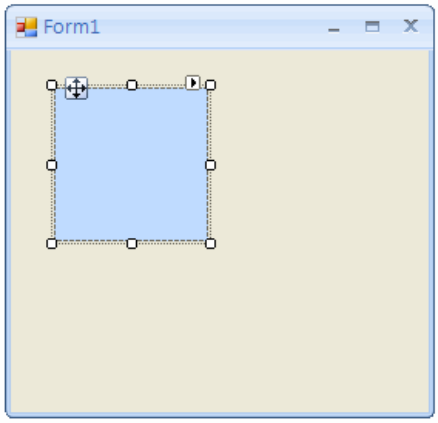
If using VB.NET then your new definition should like this: -

```
Partial Class Form1
    Inherits ComponentFactory.Krypton.Toolkit.KryptonForm
```

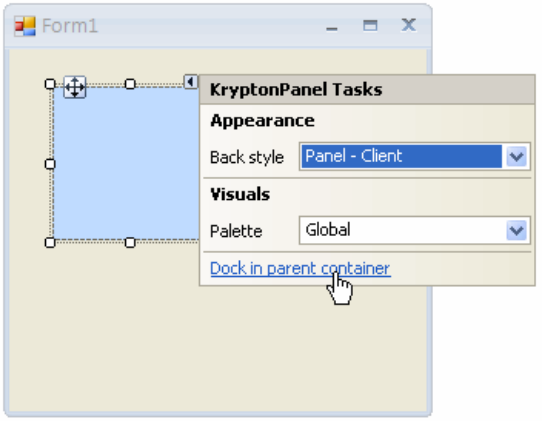
Recompile the project and then show the form in design mode again, this time you should see custom chrome applied to the form.



5) Drag a KryptonPanel from the toolbox and drop it in the centre of the form
When dropped it should look like the following picture.



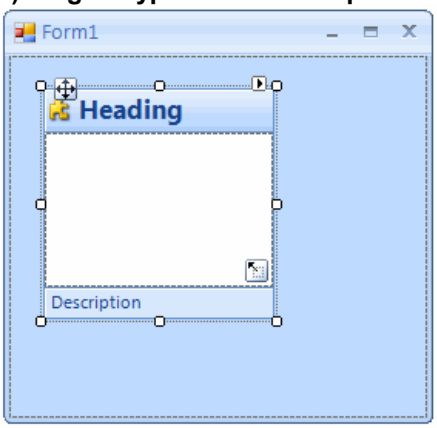
6) Use the KryptonPanel smart tag and click 'Dock in parent container'
The panel will now occupy the entire client area even when the form is resized.



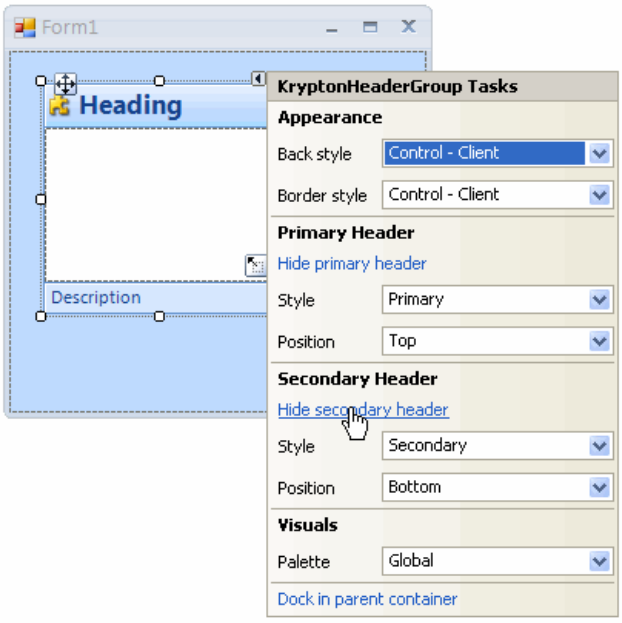
7) Modify the Padding property for the new KryptonPanel to 5 on all sides
We need to add padding because another control will be placed inside and we want a nice border around the contained control.

Padding	5, 5, 5, 5
All	5
Left	5
Top	5
Right	5
Bottom	5

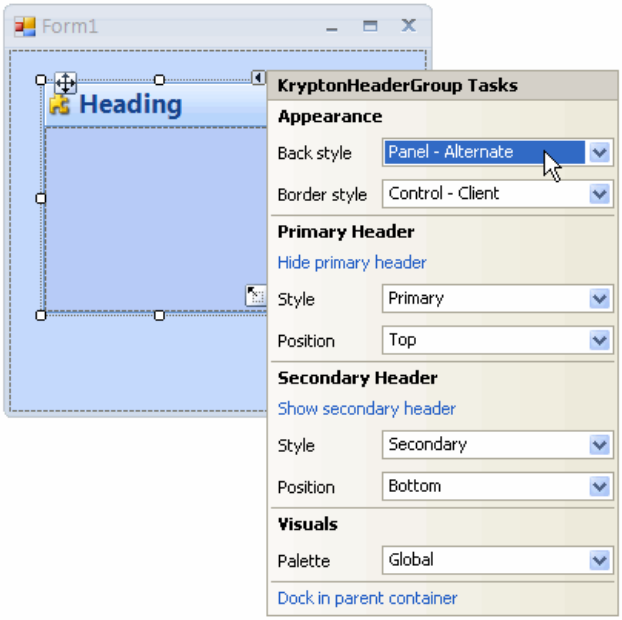
8) Drag a KryptonHeaderGroup from the toolbox and drop onto the panel



9) Use the KryptonHeaderGroup smart tag and click 'Hide secondary header'
This will result in the following picture with the header at the bottom removed from view.



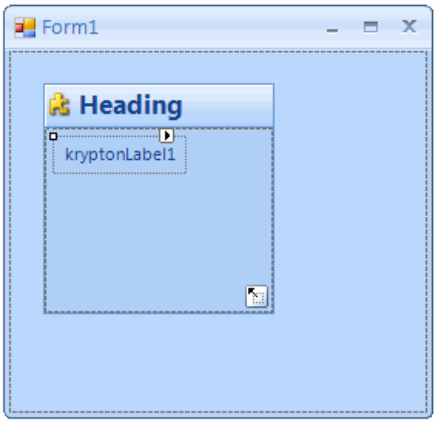
10) Use the smart tag again and change the 'BackStyle' to 'Panel - Alternate'
This gives the client area of the control the dark blue background as seen here.



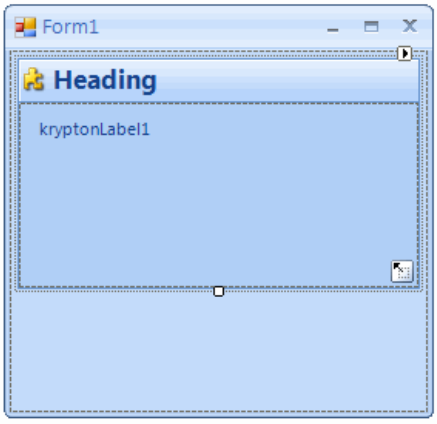
11) Click the client area to select the header groups inner panel and set Padding to be 5
By clicking in the center of the header groups client area you will change the properties window to show the properties of the contained inner panel. Once selected you then alter the padding for this inner panel to be 5 on all sides.

Padding	5, 5, 5, 5
All	5
Left	5
Top	5
Right	5
Bottom	5

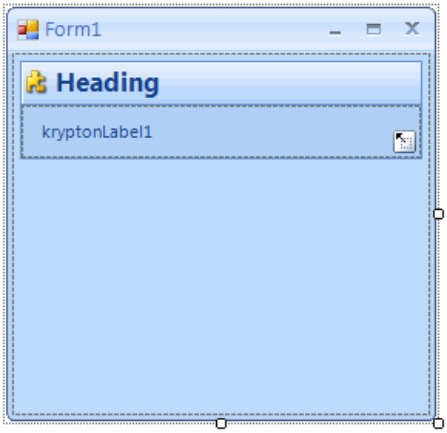
12) Drag a KryptonLabel into the client area of the KryptonHeaderGroup
Our example is going to use just a single label as the example content of the header group.



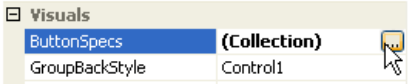
13) Select the KryptonHeaderGroup and set the 'Dock' property to 'Top'
We want the header group always placed at the top of the client area.



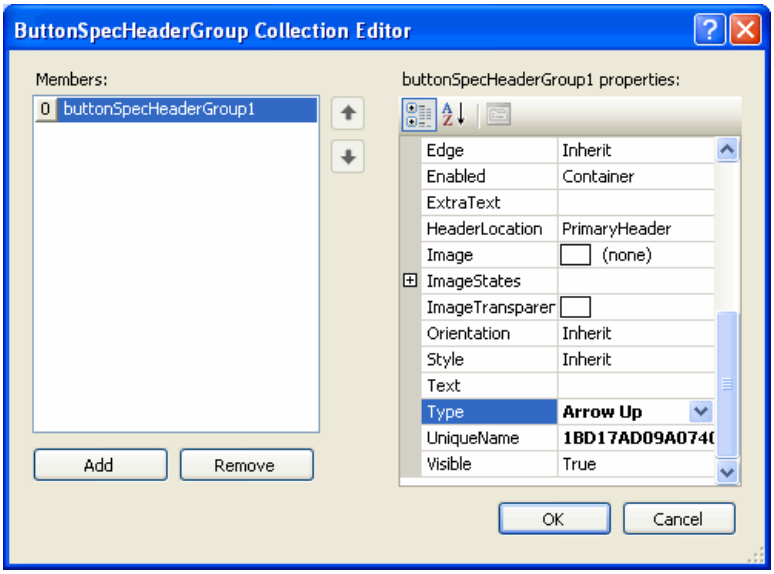
14) Now set the property 'AutoSize' to 'True'
This ensures that as the header expands and collapses the size automatically changes to reflect this. In our example the size will shrink down so that the content of the header group is shown with a 5 pixel border around it.



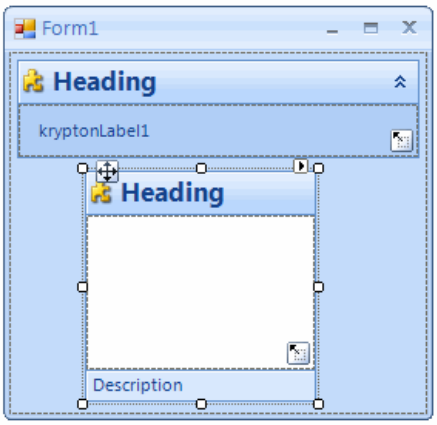
15) Find the 'ButtonSpecs' property and click the ellipses button
We we will use the collection property to define the header button we need.



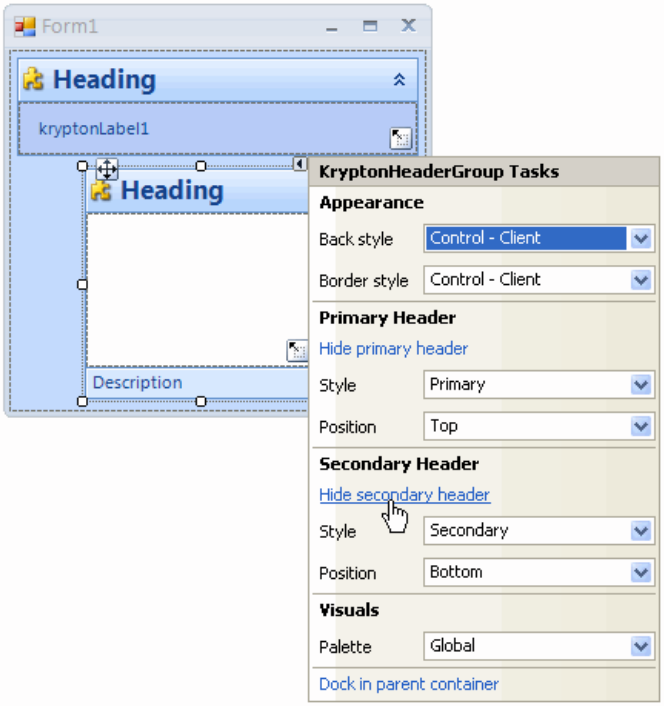
16) Add one instance to the collection and set the 'Type' to 'Arrow Up'
Use the 'Add' button on the collection editor to add a new *ButtonSpec* entry. Then set the 'Type' property of the entry to be 'Arrow Up'. The new entry should look like the following. Because the header group defaults the 'AutoCollapseArrow' property to 'True' is means then at runtime the user can press this button specification in order to toggle the expand/collapse setting automatically.



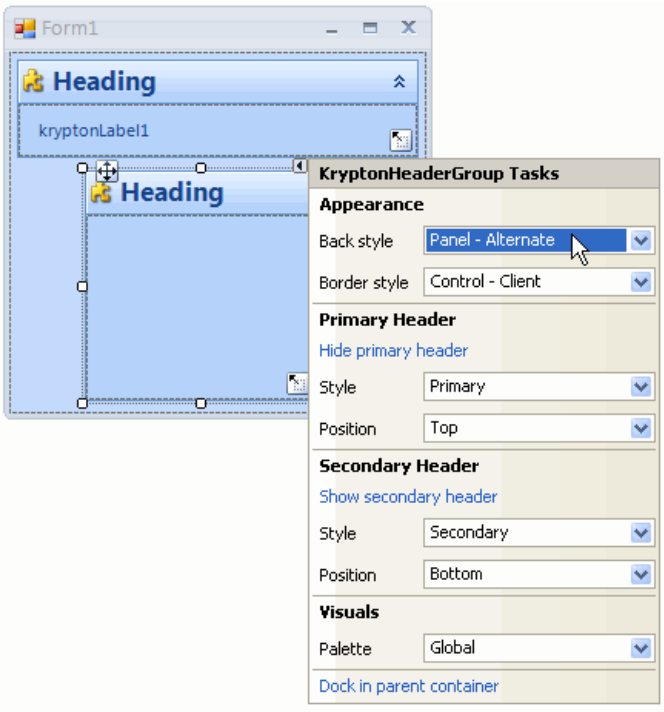
17) Drag another KryptonHeaderGroup from the toolbox and drop onto the panel



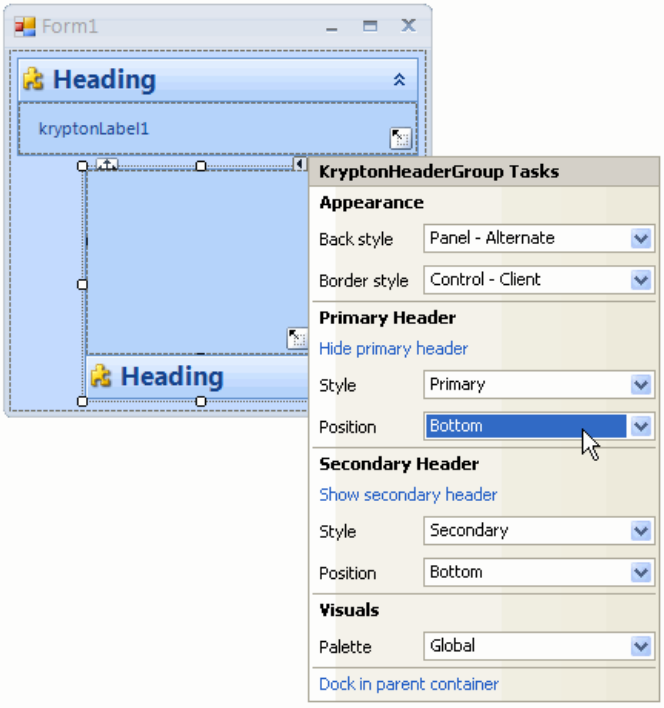
18) Use the KryptonHeaderGroup smart tag and click 'Hide secondary header'
This will result in the following picture with the header at the bottom removed from view.



19) Use the smart tag again and change the 'BackStyle' to 'Panel - Alternate'
This gives the client area of the control the dark blue background as seen here.



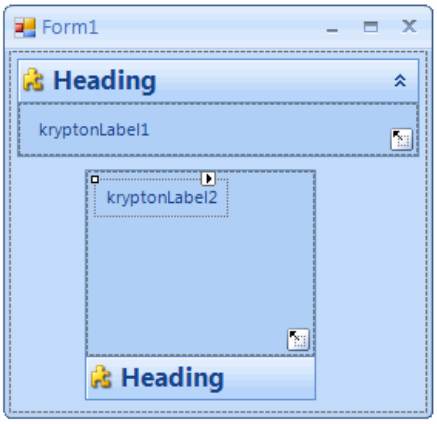
20) Use the smart tag and change the 'Position' of the 'Primary Header' to 'Bottom'
As below the primary header should be moved to the bottom of the header group control.



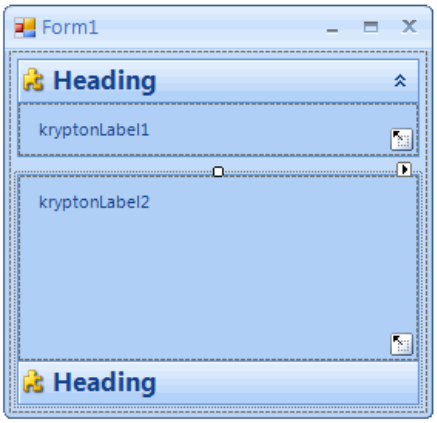
21) Click the client area to select the header groups inner panel and set Padding to be 5
By clicking in the center of the header groups client area you will change the properties window to show the properties of the contained inner panel. Once selected you then alter the padding for this inner panel to be 5 on all sides.

Padding	5, 5, 5, 5
All	5
Left	5
Top	5
Right	5
Bottom	5

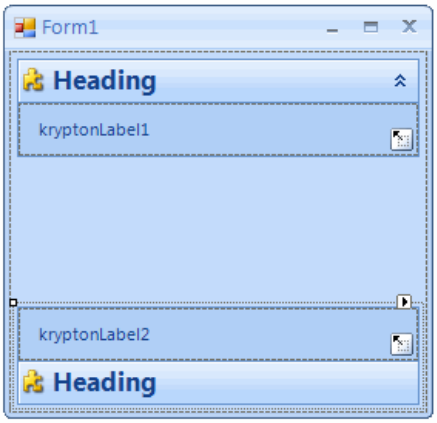
22) Drag a KryptonLabel into the client area of the KryptonHeaderGroup
Just as with the first header group, we use just a single label as the example content of the header group.



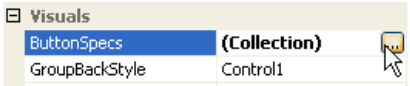
23) Select the KryptonHeaderGroup and set the 'Dock' property to 'Bottom'
We want this header group always placed at the bottom of the client area.



24) Now set the property 'AutoSize' to 'True'
This ensures that as the header expands and collapses the size automatically changes to reflect this. In our example the size will shrink down so that the content of the header group is shown with a 5 pixel border around it.

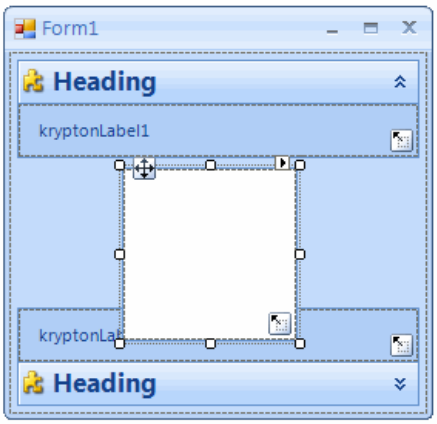


25) Find the 'ButtonSpecs' property and click the ellipses button
We we will use the collection property to define the header button we need.

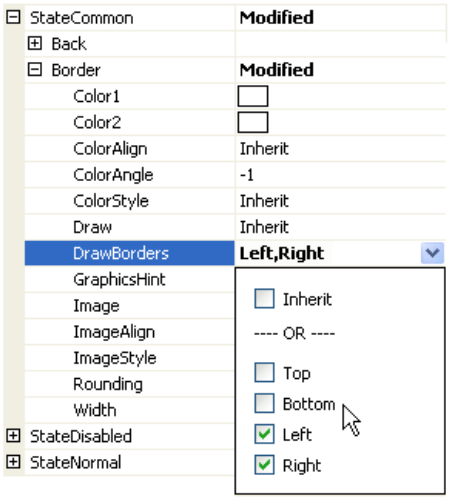


26) Add one instance to the collection and set the 'Type' to 'Arrow Up'
Use the 'Add' button on the collection editor to add a new *ButtonSpec* entry. Then set the 'Type' property of the entry to be 'Arrow Down'. Because the header group defaults the 'AutoCollapseArrow' property to 'True' is means then at runtime the user can press this button specification in order to toggle the expand/collapse setting automatically.

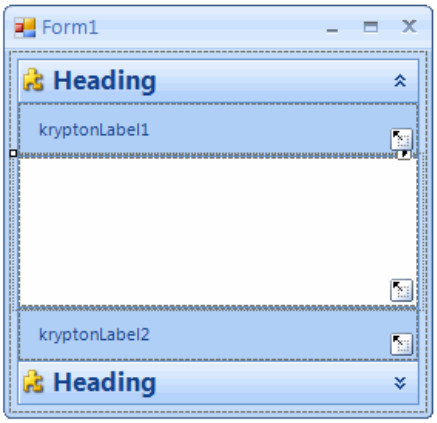
27) Drag a KryptonGroup from the toolbox and drop onto the center of the client area



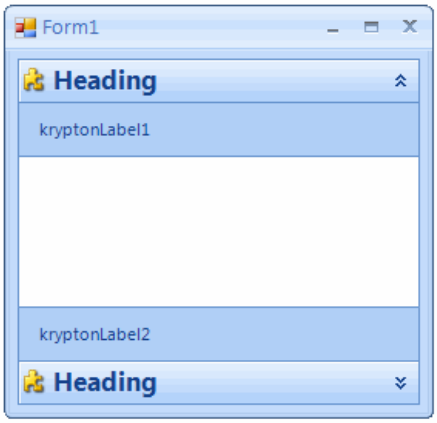
28) Update the 'StateCommon -> Border -> DrawBorders' property to 'Top,Bottom'
Because the centre group will always have a header group above and below we do not need the top and bottom borders just the let and right need drawing. The picture below shows the property that needs altering.



29) Last of all set the 'Dock' property to 'Fill'
We always want the group to fill up whatever space is left over after positioning and sizing the top and bottom header group.

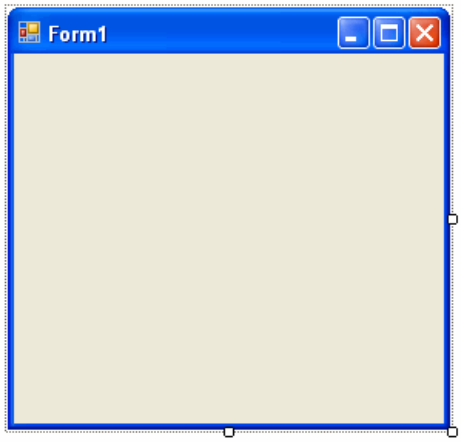


Now run the application and without writing a single line of code the top and bottom groups will collapse/expand as you click the appropriate header buttons.


Form1
Heading
kryptonLabel1
kryptonLabel2
Heading

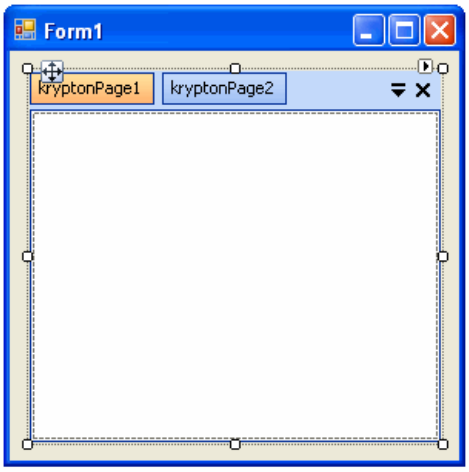
Tutorial – User Page Creation

1) Create a new Windows Forms project
This will automatically create a form in design mode as below.

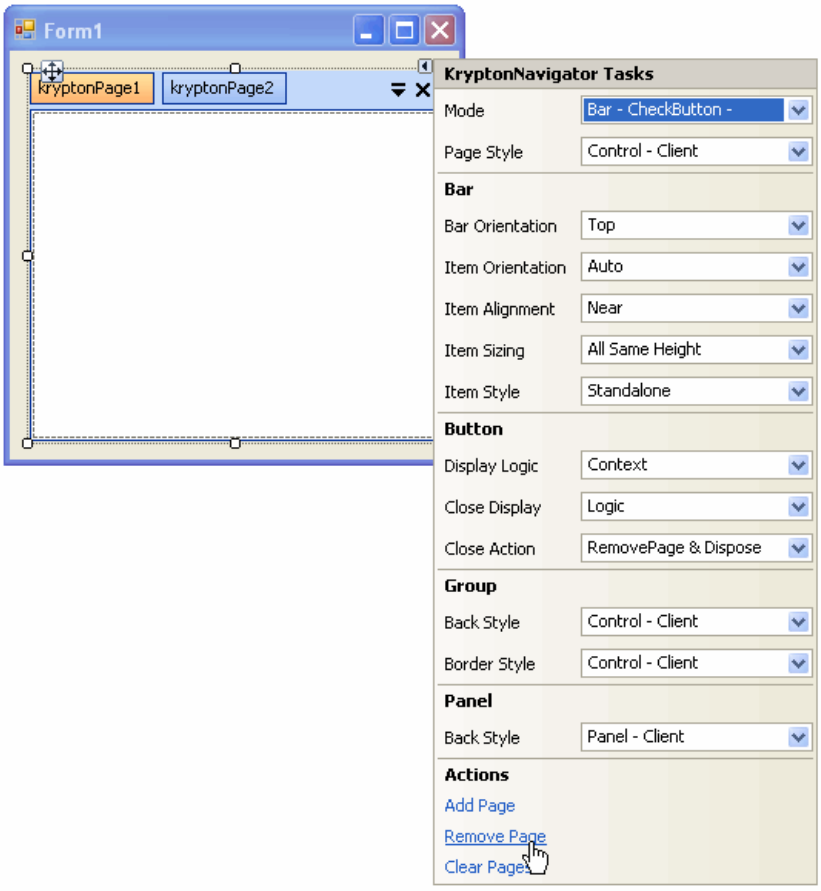


2) Ensure that the Krypton components are in the Toolbox
If not the Using Krypton in VS2005 tutorial can be used to add them.

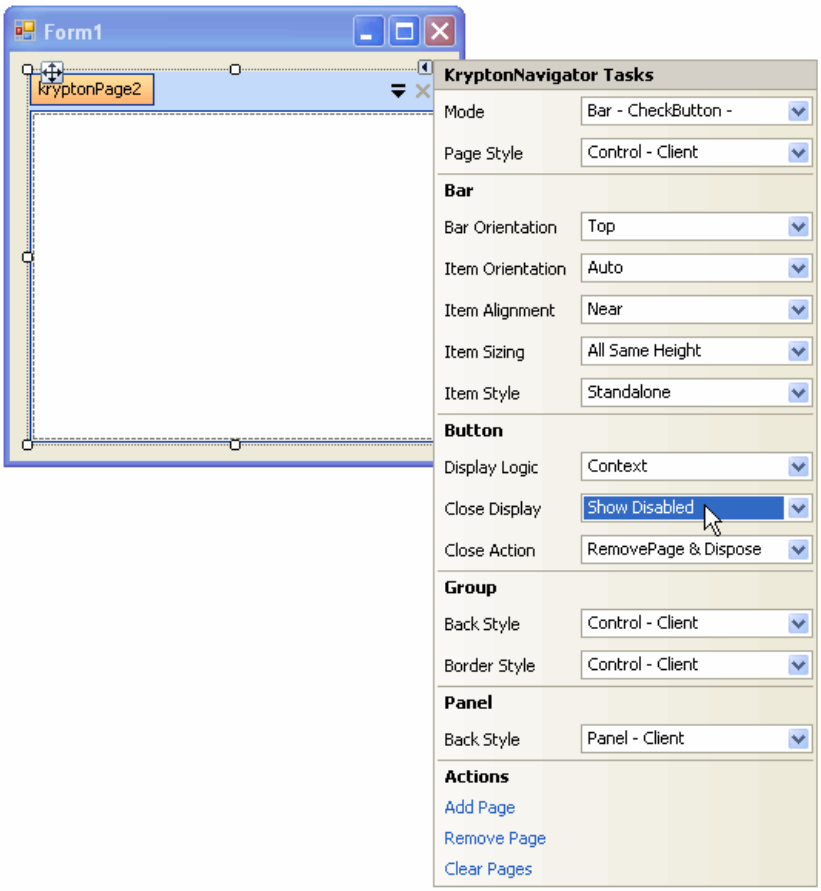
3) Drag a KryptonNavigator from the toolbox onto the client area
Once dropped resize the control so it takes up most of the client space as shown below.



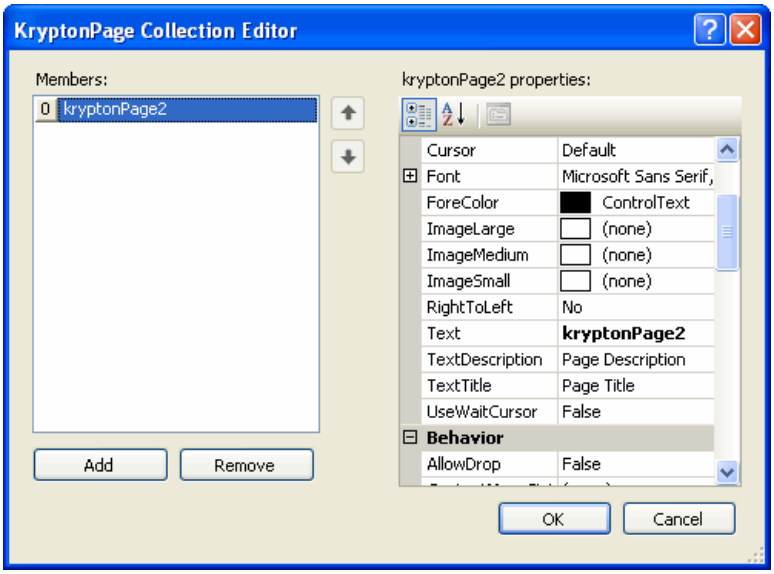
4) Click the smart tag and then select the 'Remove Page' action
We only want the navigator to have a single page at design time, so we remove one of the default pages.



5) Change the 'Close Display' property on the smart tag to be 'Show Disabled'
We want to prevent the user from deleting the single page that is now shown.



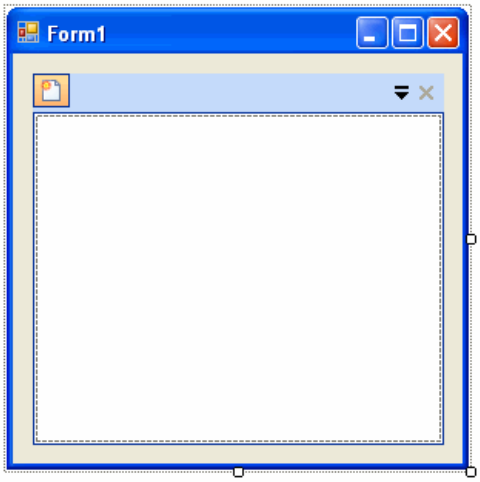
6) Click the edit button for the 'Pages' property of the navigator in the properties window
Once clicked you should have the following page collection editor displayed.



7) Set the 'Text' and 'Text Title' properties of the page to be an empty.
The page header should not display any text and so select and delete the string values for the 'Text' and 'Text Title' properties.

8) Select an appropriate new page image for the 'ImageSmall' property.
Click the 'ImageSmall' property and use the edit button to import an image for display. The example that can be run from the *Krypton Explorer* uses an image of a document with a star in the corner, to indicate that clicking the page will create a new document. You can use whatever image is appropriate for your application.

9) Press the OK button to accept the change.
You should now have the following display for the navigator at design time.



10) Double click the title bar of the form so the 'Load' event handler is generated.
We need to ensure that there is always one document page in addition to the last 'New Page' page that is present at design time. So we need to add code so that as soon as the form is loaded the initial document page is created and added to the navigator. You need to add just one line of code to the event handler so it looks like this.

```
private void Form1_Load(object sender, EventArgs e) {
    // Add the initial document page
    InsertNewPage();
}
```

11) Add the 'InsertNewPage' method implementation. Add the following simple code to create a new page add add it just before the 'New Page' entry.

```
using ComponentFactory.Krypton.Toolkit;
using ComponentFactory.Krypton.Navigator;
private void InsertNewPage() {
    // Create a new page and give it a simple name
    KryptonPage newPage = new KryptonPage();
    newPage.Text = "Page";

    // Insert page before the last page
    kryptonNavigator1.Pages.Insert(kryptonNavigator1.Pages.Count - 1, newPage);

    // Make the new page the selected page
```

```
    kryptonNavigator1.SelectedPage = newPage;

    // If this is the third page then we must have two document pages
    // and so the user is now allowed to delete document pages
    if (kryptonNavigator1.Pages.Count == 3)
        kryptonNavigator1.Button.CloseButtonDisplay = ButtonDisplay.ShowEnabled;
}
```

Once the new page has been created it is added as the second to last page. We do not insert it at the end because the 'New Page' entry is always left as the last entry. Once the page is added it is selected for use. Last of all a check is made to decide if the user is allowed to delete document pages.

12) Add a handler for the KryptonNavigator event called 'SelectedPageChanged'

Select the navigator control on the form and then use the properties window to list the available events. Find the 'SelectedPageChanged' entry and then double click the value for the property, this will cause an empty event handler to be created in the code window. Now add the following code to the handler.

```
private void kryptonNavigator1_SelectedPageChanged(object sender, EventArgs e) {
    // Selecting the 'New Page' entry should create a new page
    if (kryptonNavigator1.SelectedIndex == (kryptonNavigator1.Pages.Count - 1))        InsertNewPage();
}
```

This event is fired whenever the user selects a new page, so if they select the last page we want to perform the special action of creating a new document page for the user.

13) Add a handler for the KryptonNavigator event called 'CloseAction'

Select the navigator control on the form and then use the properties window to list the available events. Find the 'CloseAction' entry and then double click the value for the property, this will cause an empty event handler to be created in the code window. Now add the following code to the handler.

```
private void kryptonNavigator1_CloseAction(object sender, CloseActionEventArgs e) {
    // Prevent the last page from being selected when second to last page is removed
    if (e.PageIndex == (kryptonNavigator1.Pages.Count - 2))
        kryptonNavigator1.SelectedIndex = kryptonNavigator1.Pages.Count - 3;

    // Prevent the last document window from being removed
    if (kryptonNavigator1.Pages.Count == 3)
        kryptonNavigator1.Button.CloseButtonDisplay = ButtonDisplay.ShowDisabled; }
```

The purpose of the first 'if' statement is to check to see if the second to last page is being removed. If so we need to change the current selection to a previous page in order to prevent some unexpected behavior. Without this change the selection would automatically be shifted to the last page, but the last page is the 'new page' entry. When the 'new page' entry is selected it automatically creates a new page. So if we did not alter the selection manually then removing the second to last page would cause a new page to be created it its place.

The second 'if' statement check to see if the close button needs to be disabled because the close is going to leave just one document window in addition to the 'new page' entry.

14) Add a handler for the KryptonNavigator event called 'ContextAction'

Select the navigator control on the form and then use the properties window to list the available events. Find the 'ContextAction' entry and then double click the value for the property, this will cause an empty event handler to be created in the code window. Now add the following code to the handler.

```
private void kryptonNavigator1_ContextAction(object sender, ContextActionEventArgs e) {
    // Give the 'new page' entry some display text in the context menu
    e.ContextMenuStrip.Items[e.ContextMenuStrip.Items.Count - 1].Text = "New Page";
}
```

In an earlier step we removed the display text for the 'new page' page. A consequence of this is that the context menu that is displayed at runtime for selecting pages will have no text for this page entry. This looks a little confusing for users of the control and so we use this event to customize the last context menu entry with some helpful 'New Page' text.

15) Compile and run the code and you will have the following application.