Tools 2D

Components for CodeGear Vcl

Users Manual

Rt-Science

Rt-Tools2D for Vcl Users Manual - Version 3.5.0

by Horst Reichert and Edward Bradburn

Part I

1 Introduction

1.1 Welcome

RTTools2D is a compilation of components for Borland and CodeGear Delphi and C++ compilers for generating Cartesian X/Y-plots and Pie /Donut Charts from scientific and financial data. It supports a large variety of axis scaling types, series, calculated line legends, movable markers, etc., and thus gives you a tool for generating scientific and financial graphing programs easily, while retaining numerous options for tailoring the end product to the requirements of you and your customers.

In addition, labels, edits and style selection controls have been tailored to generate a modern design and user interface for your program; these work seamlessly with the properties of RTTools2D.

The first version of the components was developed mainly during the development of RTPlot. - a tool for generating Cartesian X/Y-plots from scientific data. The design and user interface of the program enables you to enter and calculate tabular data. The graphs change in real time during data entry to show the effect of your changes, and supported graph types include linear and non-linear regression, interpolation, differentiation and integration graphs. A powerful reporting module generates ready-to-publish documents. Please see http://www.rt-science.com for details.

I make no claims to perfection, neither for myself nor for my software, so if you notice any errors within this manual or the components – or if you have any wishes for future developments – please do not hesitate to contact me via e-mail (support@rt-science.com).

I wish you success in your work with RTTools2D.

October 2007 Horst Reichert

1.2 Typographical Conventions

To help you better understand the documentation, this manual uses special formatting to show you that the text has a particular meaning.

> Hints provide you with information that is non-essential to a particular topic, but which will help you work more easily or efficiently. Hints are presented inside a frame with a gray background, as here.

Keystroke descriptions use the labels as found on standard American keyboards. Depending on the manufacturer and the language of your computer, these labels may differ from those on your keyboard. Combined keystrokes, such as when pressing the Ctrl key and the C key simultaneously, are written like this: **Ctrl C**. Within the descriptions of the components the **component names** are set in bold, ***property names*** are set in bold italic and *property values* are set in italic type. Citations of Delphi and C++ `Source code` are printed using a Courier fixed-width font.

This manual extensively uses hyperlinks to refer to other topics. To assist you finding the topics in the printed manual, a small icon is displayed on the right, which points to the referenced page 3.

The pictures included in this manual are screenshots taken on a computer running CodeGear RAD Studio 2007 on Windows™ XP. These screenshots may differ to the presentation of the software on your computer if you are running a different system configuration.

1.3 Installation

RT Tools 2D was developed for Borland and CodeGear Delphi and C++ compilers. The setup file contains VCL library packages for Delphi 6 and 7, Developer Studio 2005 and 2006 and CodeGear RAD Studio 2007 and 2009, Embarcadero RAD Studio 2010, Embarcadero RAD Studio XE and Embarcadero RAD Studio XE2. VCL.net libraries are also supplied for Developer Studio 2005, 2006 and CodeGear RAD Studio 2007. Versions for Developer Studio 2006, to RAD Studio XE2 also support C++ compilers.

	VCL Win32	VCL . net	C++
Delphi 6	✓	–	–
Delphi 7	✓	–	–
Delphi 2005	✓	✓	–
Developer Studio 2006	✓	✓	✓
CodeGear RAD Studio 2007	✓	✓	✓
CodeGear RAD Studio 2009	✓	✓	✓
Embarcadero RAD Studio 2010	✓	–	✓
Embarcadero RAD Studio XE	✓	–	✓
Embarcadero RAD Studio XE2	✓	–	✓

Depending on the version you want to install please execute "Rt-Tools2D.vcl.demo.exe" (demo version), "Rt-Tools2D.vcl.std.exe" (standard full version) or "Rt-Tools2D.vcl.pro.exe" (professional full version). The setup program gives you the choice of installing for each of

the supported compilers, and you can choose whether or not you want to install the online help, example projects and Delphi sources (only professional version).

The installation program creates several main directories: “Delphi 5”, “Delphi 6”, “Delphi 7”, “Studio 2005”, “Studio 2006”, “Studio 2007”, “Studio 2009”, “Studio 2010”, “Studio XE” and “Studio XE2”. Each directory contains a complete set of the library files needed for the relevant development environment. The directories “DelphiExamples”, “CPlusPlusExamples” and “Help” contain example files plus the Help file for the respective IDE.

The installation program also extends the library path to point to the installation folders and sets up the help environment to display relevant online help inside the IDE.

After install, the components can be used straight away. To uninstall, just use the automatic Windows™ installer remove option.

After Installation to Developer Studio 2005 to RAD Studio XE2, please be patient after calling Help in the IDE firstly. The Microsoft Help System needs to update its internal structures. This takes several minutes. During this time there might be no waiting mouse cursor or update status display. This might lead you to the opinion the computer has crashed, but it will recover after some time.

The graphics library depends on the installation of GDI+. This means that the relevant gdiplus.dll for your operating system is installed. This is normally the case on Window2000 with all updates installed. Windows XP, Vista and Windows 7.

Please take care on deployment of your final product, that this is also the case on the target system.

1.4 License

This End User License Agreement ("EULA") is a legal agreement between you (either an individual or an entity) and Rt-Science for the software product identified above, which may include user documentation provided in online or electronic form. By installing, copying, or otherwise using the SOFTWARE, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install or use the SOFTWARE.

1. GRANT OF LICENCE

Rt-Science grants to you as an individual or entity a non-exclusive Licence to make and use copies of the SOFTWARE in the manner provided below. The software is licensed, not sold.

(a) Evaluation License

Rt-Science grants to you as an individual, a personal, nonexclusive license to install the SOFTWARE for the sole purposes of evaluating the SOFTWARE. You may evaluate the SOFTWARE for a period of thirty (30) days. After this period, you shall either (i) delete the SOFTWARE and all related documentation from all computers onto which it was installed or copied, or (ii) purchase a registered license from Rt-Science or one of its authorized suppliers to purchase the SOFTWARE. You may not under the terms of the evaluation license distribute any portion of the SOFTWARE or products generated using the SOFTWARE.

(b) Registered Licence

After you have purchased the licence for the SOFTWARE, and have received a fully enabled version, you are licensed to copy the SOFTWARE only into the memory of the number of computers corresponding to the number of licenses purchased and activate the SOFTWARE using the supplied installation program. The primary user of the computer on which each registered copy of the SOFTWARE is installed may make a second copy for his or her exclusive use on a second computer. Under no other circumstances may the SOFTWARE be operated at the same time on more than the number of computers for which you have paid a separate licence fee. You may not duplicate the SOFTWARE in whole or in part, except that you may make copies of the SOFTWARE for backup or archival purposes. You have the right to distribute programs (EXE files) you have created with the help of the SOFTWARE without any further license fees. You have a royalty-free right to distribute the portions of the SOFTWARE designated as "Redistributable Code" under the terms below.

(c) Source Code Licence (professional Version)

If you purchase a license for the SOFTWARE source code neither the source code nor modified source code may be distributed by you to any third party under any circumstances. The SOFTWARE source code must be protected as you would your own and you expressly and unequivocally agree to be bound by the acts of your employees and agents and the terms of the RESTRICTIONS section below. You may modify the SOFTWARE source code; however such modifications do not constitute ownership of the source code. Modifications to the SOFTWARE source code may not be sold, transferred or published in any manner whatsoever. The compiled SOFTWARE source code may only be copied to, and used on, those computers that have an installed, registered license under the terms above. You may distribute the portions of the compiled SOFTWARE source code designated as "Redistributable Code" under the terms below. Rt-Science retains all right, title and interest in and to the SOFTWARE source code. Licensing of the SOFTWARE source code does not constitute a transfer of ownership under the terms of this agreement, and the SOFTWARE remains owned and copyrighted by Rt-Science.

You acknowledge that the SOFTWARE source code is licensed "AS-IS", without warranty of any kind, and agree that Rt-Science is under no obligation to provide technical support for any original or modified SOFTWARE source code.

2. REDISTRIBUTABLE CODE

Portions of the software are designated as "Redistributable Code 42". The Software documentation describes the files and Redistributable rights associated with each file of the Redistributable Code, subject to the requirements described below. You have a royalty-free right to distribute the portions of the SOFTWARE designated as "Redistributable Code" only if:

(a) You have purchased a licence for the SOFTWARE and have received a registration code enabling the registered copy of the SOFTWARE.

(b) You distribute only the portions of the SOFTWARE designated as "Redistributable Code".

(c) You use and distribute the "Redistributable Code" only in conjunction with the binary files that make use of them as a part of your software product.

(d) You agree to indemnify, hold harmless, and defend Rt-Science and its suppliers from and against any and all claims or lawsuits including attorney's fees that arise or result from the use or distribution of your software product.

3. RESTRICTIONS

You must not redistribute the registration codes provided, neither on paper nor electronically.

You may not disassemble, decompile or reverse engineer the SOFTWARE or any portions of it.

You may not rent, lease, or lend the SOFTWARE. You may permanently transfer all of your rights under this EULA provided you transfer all copies of the SOFTWARE (including copies of all prior versions if the SOFTWARE is an upgrade) and registration codes and retain none, and the recipient agrees to the terms of this EULA.

4. TERMINATION

Without prejudice to any other rights, Rt-Science may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the SOFTWARE. You may terminate this licence at any time by destroying the original and all copies of the SOFTWARE in whatever form.

5. COPYRIGHT

The SOFTWARE is owned by Rt-Science and is protected by German copyright laws and international treaty provisions.

6. LIMITED WARRANTY

THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL THE AUTHOR or AUTHORS BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING INCIDENTAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF THE USE OF THE PROGRAM, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. YOU ACKNOWLEDGE THAT YOU HAVE READ THIS LICENCE, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US, SUPERSEDING ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS LICENCE.

7. LIMITATION OF LIABILITY

IN NO EVENT SHALL RT-SCIENCE OR ITS SUPPLIERS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES OF ANY KIND ARISING OUT OF THE DELIVERY, PERFORMANCE, OR USE OF THE SOFTWARE, EVEN IF RT-SCIENCE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY EVENT, RT-SCIENCE LIABILITY FOR ANY CLAIM, WHETHER IN CONTRACT, TORT, OR ANY OTHER THEORY OF LIABILITY, EXCEEDS THE LICENCE FEE PAID BY YOU.

Part II

2 Overview

After installation, the tool palette of the IDE will have two new categories named "Rt-Tools2D Standard" "Rt-Tools2D Graph" and "Rt-Tools2D Frames". The first contains components as edit boxes, plus style selectors that have been developed to assist you in building up the user interface of your program. They are tailored to assist you in setting up all the available properties in the Cartesian plot components. These components are stored in the second category. The third palette contains frames that can be used to built up the user interface to set the properties in the graph components with ready to use frames.

2.1 Edits, Lists and Style Selectors

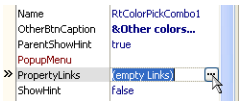
The "Rt-Tools2D Standard" category of the tool palette of the IDE contains components, namely:

- TRtRichLabel 12 - Label with enhanced styles such as bold, italic, underline, strikeout, subscript, superscript, color changes, font changes and variable display angle.
- TRtCaptionEdit 12 - Edit box for entering captions with the above enhanced styles.
- TRtIntegerEdit 13 - Input of integers with range check.
- TRtDoubleEdit 13 - Input of floating point numbers or date/time values with range check.
- TRtNumericUpDn 14 - Input of floating point numbers with up/down buttons for incrementing or decrementing the input value.
- TRtCheckBox 10 - Standard CheckBox plus PropertyLinks 9.
- TRtRadioGroup 10 - Standard RadioGroup plus PropertyLinks 9.
- TRtColorPickCombo 14 - Dropdown color selection palette.
- TRtTuneColorsDialog 15 - Replacement for the standard color dialog, enabling transparency.
- TRtFontDialog 67 - Replacement for the standard font settings dialog, enabling extended text styles.
- TRtDashStylesList 15 - List box to select line dash styles.
- TRtDashStylesCombo 15 - Combo box to select line dash styles.
- TRtAreaStylesList 15 - List box to select area hatch fill styles.
- TRtAreaStylesCombo 15 - Combo box to select area hatch fill styles.
- TRtPointSymbolsList 15 - List box to select point symbol styles.
- TRtPointSymbolsCombo 16 - Combo box to select point symbol styles.
- TRtUndoStack 16 - Component supporting undo/redo functionality
- TRtUndoButton 16 - Control giving access to the undo functionality
- TRtRedoButton 16 - Control giving access to the redo functionality

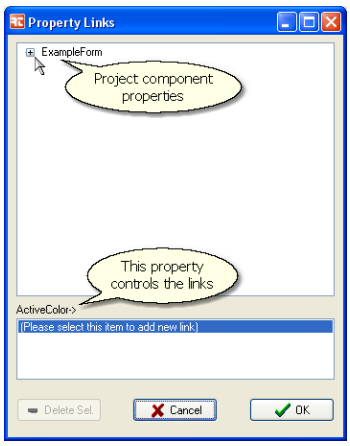
All edit and selection components can use the unique PropertyLinks 9 property to access other components.

2.1.1 Using Property Links

If you want to use an edit or selection control to set properties in another control, this usually means that you need to write an event handler – for the ***OnSelect*** event for combo boxes or the ***OnChanged*** event for edit controls, for example. Within this event method you will normally set some properties in other components, such as changing a line style in relation to an ***ItemIndex*** or providing an axis with a caption using the ***Text*** of an edit, for example. This can be very tedious, since you are simply writing down assignments of one property to other properties.

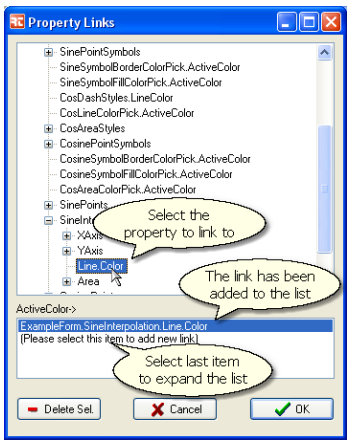


However, this same functionality can be achieved more easily using the ***PropertyLinks*** property editor. This special property editor window is opened when you click on the ellipsis button inside the object inspector.

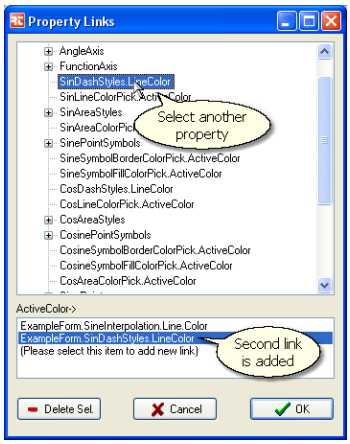


The Property Links property editor window has four major parts: the project component properties tree view at the top; a label indicating the active setting property of the calling component; a list view of existing links and new items below; and a number of control buttons.

To add a new link, select the last item in the link list at the bottom. In the Tree view above, all the available visible forms and components matching the type requirements of the selector are now shown.



You can expand the branches of the tree as you want, and select any property that you want to synchronize with the active setting property. The corresponding item in the link list will be automatically updated to match the current selection in the tree.



This procedure can be repeated for as many dependent properties as you require. In the example shown on the left, the ***ActiveColor*** property of a **TRtColorPickCombo** component will synchronize the color of a line in a graph and the color of the lines in a selection combo box. This combo is used to select the relevant line style of this line.

To delete a link, you can select it in the list below and click the Delete Sel. button. Clicking the OK button will activate the links. This means that the active setting property will receive its preset from the first item in the link list. Changing the value of the active setting property will also automatically change the linked properties without writing a line of code.

The **TRtCheckbox** component is an exact copy of the standard **TCheckBox** but adds the unique PropertyLinks 9 property of RTTools2D to set Boolean properties in the graph components that are coupled to the ***Checked*** property of the checkbox.

The **TRtRadioGroup** component is an exact copy of the standard **TRadioGroup** but adds the unique PropertyLinks 9 property of RTTools2D to set Integer properties in the graph components that are coupled to the ***ItemIndex*** property of the radio group.

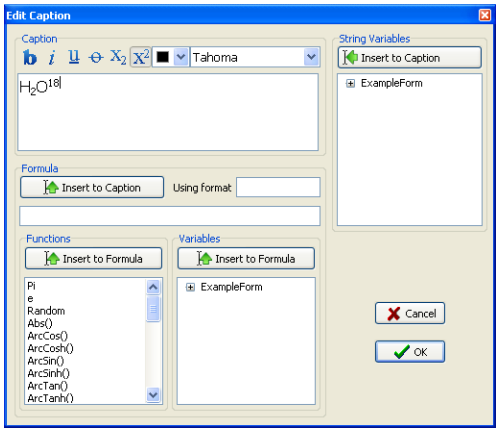
2.1.2 Using Captions with Enhanced Styles

If you are a scientist then you will need advanced captioning facilities for your graphs in order to present your captions properly. RTTools2D gives you access to the whole range of Windows font capabilities by using Wide Strings for Compilers previous to Studio 2009 and Unicode Strings for Studio 2009 as an internal type for **TRtCaption**. This means that you can use any 16-bit character provided by the current font.

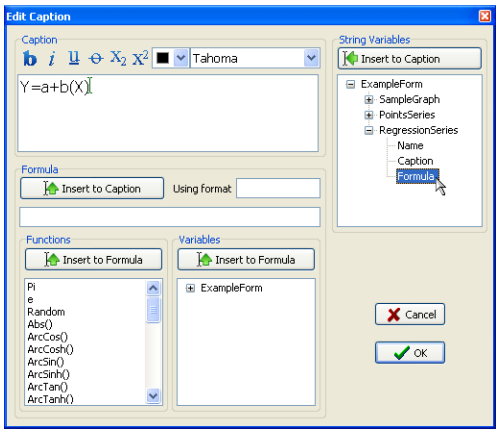
In addition, formatting styles as bold, italic, underline, strikeout, subscript and superscript are supported. You can change the color and current font within the caption text itself. You can also insert string variable references and numerical formula results into the text. These variables will be automatically updated if the related component properties change.

The style changes are controlled by special codes 70, as described in the reference. Since it may well be very tedious to remember all these codes and enter them manually, a special property editor is available to help you alter your captions:

To open the captions property editor, click on the ellipsis button with the relevant caption property in the object inspector.



The editing window is separated into three major groups. The "Caption" group contains the edit box for the caption and a toolbar showing the styles, color and font of the current selection. The style can be changed by toggling the buttons and changing the selections in the color and fonts dropdown controls.



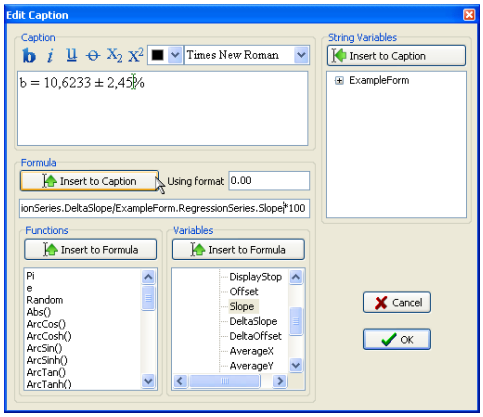
The "String Variables" group on the right displays a tree of all forms containing any RTTools2D components that return automatically-updated string variables. If you select a property in this tree and click the “Insert to Caption” button or double-click the property, the variable will be included at the insert position in the caption edit box. You can also drag & drop the property to any position in the caption.

The "Formula" group enables you to enter a formula of your choosing. The formula may

contain a number of operators that are compatible with Delphi or C++ syntax, plus the power operator, ^. Case is not significant when matching function names or the 'E' character used in scientific notation. Spaces and tabs are ignored.
The grammar follows the normal rules of arithmetic precedence, with ^ highest, * and / in the middle, and + and – lowest. Thus 1+2*3^4 is equivalent to 1+(2*(3^4)). Note that the power operator X^y is right-associative: 2^0.5^2 is thus equivalent to 2^(0.5^2). All other arithmetic operators are left-associative: 1-2-3 is equivalent to (1-2)-3. Parentheses can be used to force non-default grouping. Trigonometric functions such as sin(), cos(), etc., have the radiant as the argument, while the reciprocal angle functions have the radiant as the result.

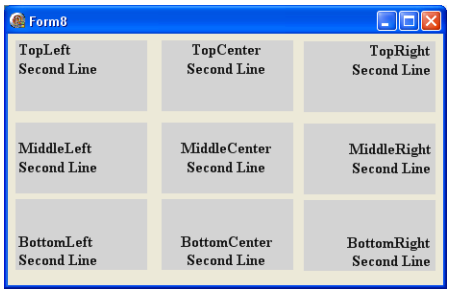
Table: *Syntax for Formulas*

Variables	
	Fully qualified name of the component + "." + name of the result property
Operators	
+	Plus
-	Minus
*	Multiply by
/	Divide by
^	To the power of
Functions	
abs()	Absolute value
arccos()	Inverse cosine
arccosh()	Inverse hyperbolic cosine
arcsin()	Inverse sine
arcsinh()	Inverse hyperbolic sine
arctan()	Arctangent
arctanh()	Inverse hyperbolic tangent
ceil()	Lowest integer greater than or equal to argument. The absolute value of the argument must be less than 2147483647.
cos()	Cosine
cosh()	Hyperbolic cosine
cotan()	Cotangent
DegToRad ()	Degrees to radiant (same as: *pi/180)
exp()	Exponential (power to base e)
floor()	Highest integer less than or equal to argument. The absolute value of the argument must be less than 2147483647.
frac()	Fractional part
int()	Integer part
ln()	Natural logarithm
lg(), log10()	Logarithm with base 10
log2()	Logarithm with base 2
RadToDeg ()	Radiant to degrees
round()	Argument rounded to the nearest whole number.
sin()	Sine
sinh()	Hyperbolic sine
sqr()	Square
sqrt()	Square root
tan()	Tangent
tanh()	Hyperbolic tangent
Constants	
pi	3.14159…
e	2.71828...

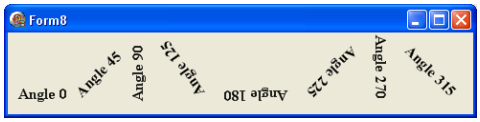


To assist you in writing the formula, two subgroups are available: "Functions" lists all numerical functions, while "Variables" lists all numerical calculation results available. To insert the function or variable into the formula, simply double-click the item or select and click the appropriate “Insert to Formula” button. Lastly, the "Using Format" field can be used to enter an optional format string that will control the format of the formula when inserted into the caption. The field uses the same syntax as the ***FormatFloat*** function. The items of both lists can be dragged and dropped to any position in the formula edit field. Variables can also be dragged to the caption edit box.

As described above, the **TRtRichLabel** control has a ***Caption*** Caption property that enables enhanced formatting options.



Although the label has no word-wrap property, it can display multi-line captions if they contain line breaks (#13). The ***TextAlign*** property will set the line positions relative to the control. The example on the left demonstrates these settings (AutoSize property is set to *false*).



The ***Angle*** property controls the angle of the caption text. It can be set to any value between 0° and 360°.

If you want the user to be able to enter and change captions with the above enhanced styles, then you need to add a **TRtCaptionEdit** control to a form.

The control will work as a standard multi-line **TRichEdit**. The use of this control to set enhanced styles for a **TRtRichLabel** is demonstrated in the "CaptionEditor" example program:
The checked state of the style toolbar buttons, color and font selections is set in the ***OnSelectionChange*** event. The states of the ***Bold***...***SuperScript*** buttons are set to the Boolean states of the corresponding control properties. The color is set to the ***SelColor*** of the control and the font selection to the current ***FontName***.

This control can use PropertyLinks 9 to access properties of other components via its ***Caption*** property.

Table: *Keys active in the RtCaptionEdit*

Key	Function
←	Move caret one position left
→	Move caret one position right
Ctrl ←	Move caret to start of word
Ctrl →	Move caret to start of next word
Home	Move caret to start of input
End	Move caret to end of input
Shift ←	Extend selection one position left
Shift →	Extend selection one position right
Ctrl Shift ←	Extend selection to start of word
Ctrl Shift →	Extend selection to start of next word
Shift Home	Extend selection to start of input
Shift End	Extend selection to end of input
(Backspace) ◄	Delete character left of caret / delete selection
Del	Delete character at caret right / delete selection
Ins	Switch insert/overwrite mode
Ctrl Del	Erase from caret to end of input
Ctrl A	Select all
Esc Ctrl Z	Undo changes
Ctrl C	Copy selection to clipboard
Ctrl X	Cut selection to clipboard
Ctrl V	Paste text from clipboard
Ctrl +	Superscript mode on/off
Ctrl -	Subscript mode on/off
Ctrl B	Bold mode on/off
Ctrl I	Italic mode on/off
Ctrl U	Underline mode on/off
Ctrl O	Strikeout mode on/off

2.1.3 Numerical Input

The **TRtIntegerEdit** control is a text box specially designed for integer input.

If you specify ***MaxValue<>MinValue***, every change of the control will be checked against this range and the ***Value*** property will only change if it is within a valid range. If the entered value is wrong, a short beep will sound. The ***OnValueChanged*** event is triggered every time the value changes.

This control can use PropertyLinks 9 to access properties of other components via its ***Value*** property.

The **TRtDoubleEdit** control is specially designed for the input of floating point numbers and date/time values.

If you specify ***MaxValue***<>***MinValue***, every change of the control will be checked against this range and the ***Value*** property will only change if it is within a valid range. If the entered value is wrong, a short beep will sound. The ***OnValueChanged*** event is triggered every time the value changes.

The ***AsDateTime*** property specifies whether the Value is displayed as a date/time value or as a floating point number. This can be helpful to edit fields such as the start and stop ranges of axes, which can be either normal or date/time scales.

If ***AsDateTime*** is set to *false* and the normal floating point display is selected, then you can use the ***Format*** string to modify the display. The ***Format*** string is interpreted in the same way as with the ***FormatFloat***function. A value of "1.234" with a format of "0.00" will display "1.23" when the edit box does not have the focus. The internal value however remains unchanged and will be available when the control is again activated. If the number does not fit in the edit box, it is rounded down until the decimal separator is covered by the right border. The string "###" indicates that the number is too large to fit the integer part into the edit box.

This control can use PropertyLinks 9 to access properties of other components via its ***Value*** property.

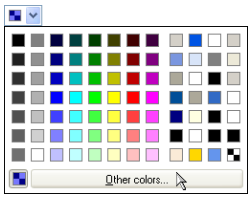
The **TRtNumericUpDn** control is specially designed for the input of floating point numbers and can increment and decrement numbers using up/down buttons.

As with the numerical editing controls noted above, this control also supports range checking using ***MaxValue***<>***MinValue***. The number of decimal places of the number that are displayed can be set with the ***DecimalPlaces*** property. The number is incremented or decremented by the value specified in the ***Increment*** property when the up/down buttons are used. The ***OnValueChanged*** event is triggered every time the value changes.

This control can use PropertyLinks 9 to access properties of other components via its ***Value*** property.

2.1.4 Style Selectors

The **TRtColorPickCombo** control can be used to set color values. This dropdown control will open a color pick palette when clicked.



The currently selected color can be accessed via the ***ActiveColor*** property, which is shown as a depressed button. The left part of the palette is larger and shows different shades of the major colors. On the right, some colors from the Windows system settings are listed. If you place the mouse pointer over a palette item for a few seconds, a tooltip will appear, showing a description such as "Button Face" or "Window", for example.

As the Cartesian plot components support transparency you may select any Alpha-RGB color. To visualize the transparency effect, the color samples are presented against a black and white checkered background.

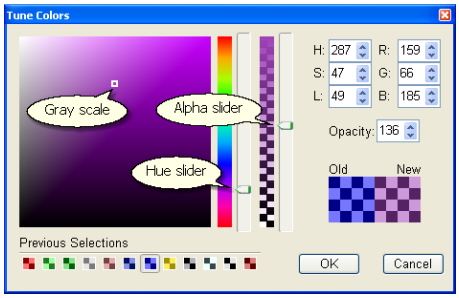
You may need to select a color that is unavailable within the color pick palette. In this case, you can open the tune colors dialog 15 using the “Other colors...” button or Ctrl clicking any palette color to modify this color. Using this dialog you can set up any Alpha-RGB color value desired. When you re-open the color picker, this new "other" color will be shown in the left bottom corner.

If you set ***DirectlyShowTuneColors*** to true, clicking the dropdown control will not show the color pick palette but will directly open the tune colors dialog.

> If you want to use this control to set colors other than the Alpha-RGB **TRtColor** used by the Cartesian plot components, you can use the ***AsTColor*** property instead. This will automatically convert from and to the correct **TColor** values. Remember to set ***EnableAlpha*** to *false* in such cases.

This control can use PropertyLinks 9 to access properties of other components via its ***ActiveColor*** property.

The **TRtTuneColorsDialog** component is a replacement for the standard color dialog; it is provided because it is necessary to adjust the Alpha-RGB color values used with the Cartesian plot controls.

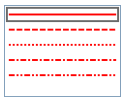


The currently selected color can be accessed via the ***ActiveColor*** property. The square at the left shows all the shades of gray using the currently selected color hue. A small frame shows the current color selection. In the middle, there are two sliders. One slider represents the rainbow colors: the slider position will set the current hue value of the color. Next to it is a slider representing the transparency (Alpha) of the color. Using the H, S, L numerical up/down controls you can set the values for hue, saturation and luminance.

The R, G, B and Opacity controls will set the Red, Green, Blue and Alpha components of the color value.
To visualize the transparency effect, the color samples are presented against a black and white checkered background.
You can restore the previous color settings by clicking the relevant icon at the bottom of the screen.

This control can use PropertyLinks 9 to access properties of other components via its ***ActiveColor*** property.

The **TRtDashStylesList** control shows a selection list of all available line styles – such as solid, dashed, dotted, etc.. The ***DashStyle*** property will give you access to the currently selected style.

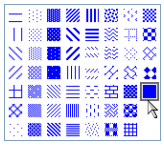


This control can use PropertyLinks 9 to access properties of other components via its ***DashStyle*** property.

The **TRtDashStylesCombo** control presents a dropdown list of all available line styles – such as solid, dashed, dotted, etc.. ready for selection. The ***DashStyle*** property will give you access to the currently selected style.

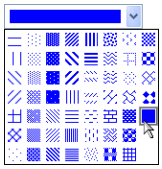
This control can use PropertyLinks 9 to access properties of other components via its ***DashStyle*** property.

The **TRtAreaStylesList** control shows a selection list of all available area hatch styles. The ***AreaStyle*** property will give you access to the currently selected style.



This control can use PropertyLinks 9 to access properties of other components via its ***AreaStyle*** property.

The **TRtAreaStylesCombo** control presents a dropdown list of all available area hatch styles. The ***AreaStyle*** property will give you access to the currently selected style.



This control can use PropertyLinks 9 to access properties of other components via its ***AreaStyle*** property.

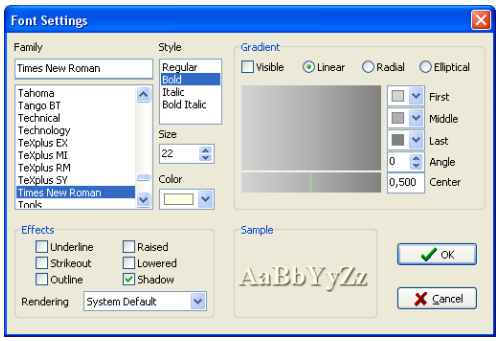
The **TRtPointSymbolsList** control presents a list of all available point symbols. The ***PointSymbol*** property will give you access to the currently selected style.

This control can use PropertyLinks 9 to access properties of other components via its ***PointSymbol*** property.

The **TRtPointSymbolsCombo** control presents a dropdown list of all available point symbols. The ***PointSymbol*** property will give you access to the currently selected style.

This control can use PropertyLinks 9 to access properties of other components via its ***PointSymbol*** property.

The **TRtFontDialog** component is a replacement for the standard font dialog; it is provided because RtTools2D gives enhanced capabilities drawing text as drop shadow, raised, lowered outline effects gradient fill etc.



The current ***Font*** property gives access to all the relevant settings as font name, style, drawing effects, size fill and gradient colors.

A sample always shows the result of the currently selected options.

You can hide the gradient settings switching ***ShowGradient*** to *false*.

2.1.5 Undo Support

The **TRtUndoStack** component implements undo/redo functionality to your programs.

In most modern programs, user actions can be undone or redone by using an appropriate menu function, and RtTools2D follows this approach: components related to Cartesian graphs have built-in support for using the undo stack. The only requirement is to set the ***UndoStack*** property of a **TRtGraph2D** control. Changes to any published property of the graph, axes, series, legend or markers are pushed onto the stack, ready to be undone by the ***Undo*** method of the stack. Any changes undone can be replayed by the ***Redo*** method of the stack.

Using the undo function with other components is comparably simple: you merely need to use the ***StartGroup***, ***PushProperty*** and ***StopGroup*** methods to store the undo states.

To undo the last stored operation, simply call the ***Undo*** method; the index of the required level can be added as the parameter to restore earlier operations. During the undo process, the state of the properties to be changed is stored in an internal redo stack. You can thus replay the previous changes using the ***Redo*** method.

The **TRtUndoButton** control is a specialized 'shortcut' button supporting undo functionality.

The control can be placed anywhere on a form or a toolbar. It supports all modern themed flat button styles. Clicking the left side of the button will execute the ***Undo*** method of the **UndoStack** to a depth of one level. Clicking on the arrow will show a dropdown, listing deeper undo levels that can be selected.

The **TRtRedoButton** control is a specialized 'shortcut' button supporting redo functionality.

The control can be placed anywhere on a form or a toolbar. It supports all modern themed flat button styles. Clicking the left side of the button will execute the ***Redo*** method of the **UndoStack** to a depth of one level. Clicking on the arrow will show a dropdown, listing deeper redo levels that can be selected.

2.2 Plot Components

The "Rt-Tools2D Graph" category of the tool palette of the IDE contains components and controls for creating Cartesian plots:

TRtGraph2D 23 - The Cartesian graph control that contains the axes, series, legends, etc.

TRtAxis 24 - Add as many as you need to the left, right, bottom or top of the graph.

TRtLegend 28 - Displays the series styles and captions: add to the graph or to any other place on the form.

TRtGraphSettingsTool 36 - Provides a tool window for setting all possible options for the graph.

TRtDoubleVector 21 - Component to store the data in a dynamically growing array or link to a database numerical field - used as source for the series data points.

TRtLineSeries 27 - Series components for showing lines, with optional filled areas beneath them.

TRtPointSeries 28 - As above, but with point symbols.

TRtPointWithErrorSeries 28 - As above, but with additional error indicators.

TRtBars 28 - Series component for showing bars as rectangles, cuboids or cylinders. They can be shifted or stacked, and shown both vertically and horizontally.

TRtBubbles 29 - Series for showing bubbles, with the radius representing a particular range.

TRtArrows 29 - Series for displaying 2D vector fields as arrows.

TRtOHLC 29 - Series for displaying financial charts, with opening, highest, lowest and closing prices.

TRtCandleSticks 29 - As above, but with bars indicating the opening and closing price range, shadows giving the highest/lowest range and color indicating rising or falling prices.

TRtLinearRegression 30 - Straight line linear regression: $Y = a + bX$.

TRtGeneralLinearLeastSquare 30 - Linear least squares fit to any function with linear dependent coefficients a_i: $Y = \sum_i a_i f_i(X)$

TRtPolynomial 31 - Polynomial regression: $Y = a + bX + cX^2$...

TRtFittedLine 125 - Nonlinear regression for any function $Y = f(X)$, given the appropriate starting values for a, b... The formula can be either supplied as a method or entered as string and solved by a built-in parser.

TRtMovingAverage 33 - Moving average line. Often used in trend analysis in financial charts.

TRtInterpolation 34 - Interpolating line. Calculation method as Akima, approximative spline, normal cubic spline smoothing, etc. is selectable.

TRtDifferential 34 - Differential of the interpolated line.

TRtIntegral 129 - Integral of the interpolated line.

TRtDiscreteFourier 35 - Discrete Fourier series of the source data.

TRtVerticalMarker 37 - Vertical line that can be moved using the mouse to indicate X

values, with an optional caption.


TRtHorizontalMarker 37 - Horizontal line that can be moved using the mouse to indicate Y values, with an optional caption.


TRtCrossHair 37 - Crosshair that can be moved using the mouse to indicate (X, Y) positions, with an optional caption.


TRtLabelWithArrowMarker 37 - A marker with a caption and an arrow pointing to positions on the graph – such as peaks, for example.


TRtPieChart 38 - Pie and Donut Charting control.


TRtFillPalette 38 - Component to store a fill styles palette for the pie segments.

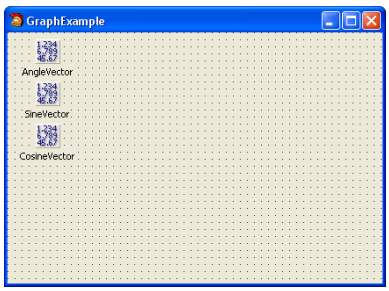

TRtPieLegend 39 - Displays the pie segment styles and captions: add to the pie chart or to any other place on the form.


TRtPieSettingsTool 40 - Provides a tool window for setting all possible options for the pie chart.

2.2.1 Example Usage

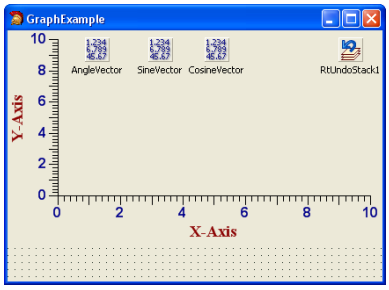
The task of generating a two-dimensional graph may initially seem complicated when you look at the Cartesian graph components and their numerous options. A walkthrough of a small example program is thus offered as a usage demonstration. The sample program draws the sine and cosine functions in the range 0-360° as 0-2π:

First, the data to be generated is assessed: the X-data values of the sine and cosine functions will be the same, while sine and cosine will have different function (Y) values for different values of X.

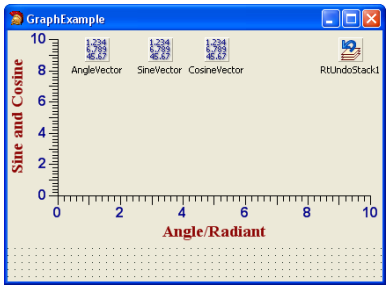


Now create a new project in the IDE and name the form GraphExample. Open the "Rt-Tools2D Graph" palette and place three **TRtDoubleVector** components on the form. Using the object inspector, rename these to *AngleVector*, *SineVector* and *CosineVector*. Save the form as "ExampleGraph" and the project as "GraphTest".

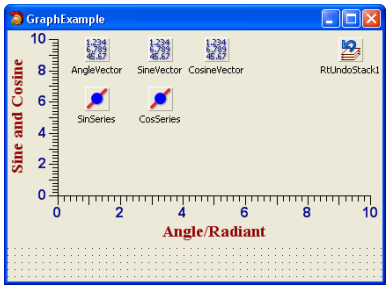
> The data vectors can be used to store any numerical data as floating point (**double**) numbers or **TDateTime** values. You can also use them without the Cartesian plot components, for storing data as an array or a list of doubles whose size is automatically adjusted.



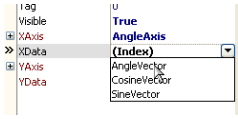
Now place a **TRtGraph2D** component on the form and adjust it to fit. On creation, the graph contains two axes: XAxis1 (with ***Caption*** *X-Axis)* and *YAxis1* (with ***Caption*** *Y-Axis)*.



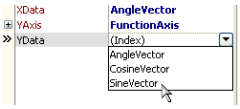
Now change the names and captions of the axes. The x-axis is modified by clicking on it and editing the corresponding ***Name*** to read *AngleAxis* and the ***Caption*** to read *Angle/ Radiant* within the object inspector. Do the same for the y-axis, setting ***Name*** to *FunctionAxis* and ***Caption*** to *Sine and Cosine*.



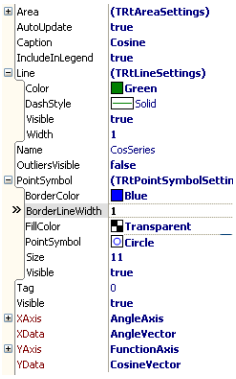
With the graph now visible, add data points to display on it to demonstrate the sine and cosine functions. To do this, place two **TRtPointSeries** components on the form. Set their names to *SinSeries* and *CosSeries* and the captions to Sine and Cosine, respectively.



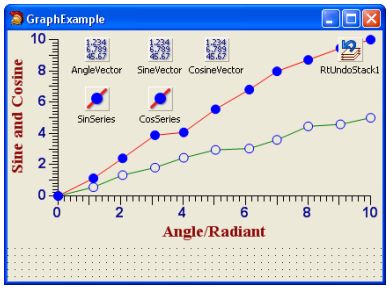
Within the form, select both series components. In the object inspector, set the ***XData*** property to *AngleVector*.



Select the **SinSeries** component and set the ***YData*** property to *SineVector*.



Analogously, select the **CosSeries** component and set the ***YData*** property to *CosineVector*. Since you will want to have different symbols and lines for the series, set the ***Line.Color*** to *Green* and the ***PointSymbol.FillColor*** to *Transparent*.



To assist you in finding the most appropriate line and point settings, the graph will show some random points representing each series. Your graph should now look similar to the graph displayed on the left.

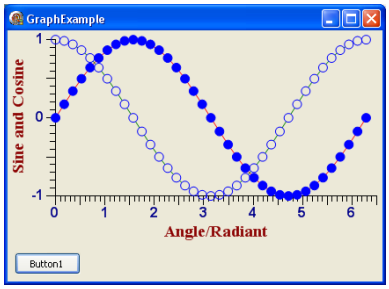
Since with this example you want to display the sine and cosine functions, you need to implement a method to fill the data vectors mentioned above. The simplest way to do this is by placing a **TButton** component to the form and then setting the ***OnClick*** event to the following procedure:

Delphi:

```
procedure TGraphExample.Button1Click(Sender: TObject);
// adds sine and cosine function values every 10°
var i, NextIdx: integer; AngleAsRad: Double;
begin
  for i := 0 to 36 do
  begin
    // using NextIdx instead of i. Clicking twice will extend graph
    NextIdx := AngleVector.Count;
    AngleAsRad := DegToRad(NextIdx*10);
    AngleVector[NextIdx] := AngleAsRad;
    SineVector[NextIdx] := Sin(AngleAsRad);
    CosineVector[NextIdx] := Cos(AngleAsRad);
  end;
end;
```

C++:

```
void __fastcall TgraphExample::Button1Click(TObject *Sender)
// adds sine and cosine function values every 10°
{
   for ( int i = 0; i <= 36; i++ )
   {
      // using nextIdx instead of i. Clicking twice will extend graph
      int nextIdx = angleVector->Count;
      double angleAsRad = DegToRad(nextIdx*10);
      angleVector->Items[nextIdx] = angleAsRad;
      sineVector->Items[nextIdx] = Sin(angleAsRad);
      cosineVector->Items[nextIdx] = Cos(angleAsRad);
   }
}
```



As you see, the data vectors can be handled as if they are zero based arrays of double. But writing to it, you must not take care for the size of the array. This is adjusted automatically by the **TRtDoubleVector** component.

If you want to compile the project and run it, please click "Button1". Your output should look like the image on the left.

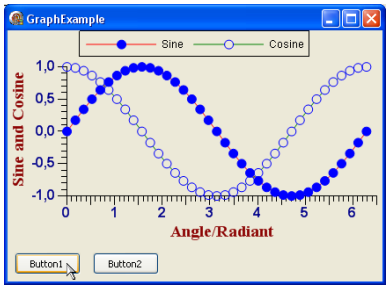
Another way to access the data is to treat it as a list of **doubles**. This is described in the example with the ***OnClick*** event of a second button. On clicking this button, the graph is extended by one point.

Delphi:

```
procedure TGraphExample.Button2Click(Sender: TObject);
// adds additional points 10° after the last
var AngleAsRad: Double;
begin
  AngleAsRad := DegToRad(AngleVector.Count*10);
  AngleVector.Add(AngleAsRad);
  SineVector.Add(Sin(AngleAsRad));
  CosineVector.Add(Cos(AngleAsRad));
end;
```

C++:

```
void __fastcall TgraphExample::Button2Click(TObject *Sender)
// adds additional points 10° after the last
{
   double angleAsRad = DegToRad(angleVector->Count*10);
   angleVector->Add(angleAsRad);
   sineVector->Add(Sin(angleAsRad));
   cosineVector->Add(Cos(angleAsRad));
}
```



You might want a legend for the series, showing line styles and captions. Do this by adding a **TRtLegend** component to the graph. Set the ***Position*** property to *TopCenterOfGraph* then you can generate the output shown on the left.

As you can see, it takes only a few minutes to create a fully working graph program. The following chapters describe the use of the above components that produce the Cartesian plot in a more detailed way.

2.2.2 Data Storage

The **TRtDoubleVector** component is specially designed for the storage of data in a dynamically growing array or as links to a numerical database field.

As described in the example above, this component is used to store the numerical data. The internal data are stored in an array of ***doubles***. The component enables access to the values via its default property Items. Thus it can be accessed as if it were an array. In addition, you can use standard **TList** methods to access the data as ***Add(), Insert(), Delete (),*** etc., to access the data. The size of the internal array is automatically adjusted to the size needed, although you can specify the ***Capacity*** to pre-set the size as required. This might be useful if you expect large amounts of data and want to prevent memory fragmentation.
Although the data is stored internally as an array of doubles, you can also use it to store date/time values because the Delphi ***TDateTime*** type is equivalent. Thus you can use the vectors that also display series using a date/time scaled x-axis.

Database Access

The component can be also set up to be database-aware. While it is fairly unusual for VCL components to support both direct data and database-linked access, it is however common for .NET WinForms components. In our opinion this simplifies matters, since you do not need to add any additional Tdb... components to the palette. To use the **TRtDoubleVector** component as database link, select the relevant ***DataSource*** the database is connected to and use the ***DataField*** name to display the values. If a series has been assigned ***XData*** or ***YData*** vectors with the database properties set, a change in the database will be reflected in the graph display. This is demonstrated in the "VectorsFromDatabase" example that is found in the "DatabaseAccess" directory.

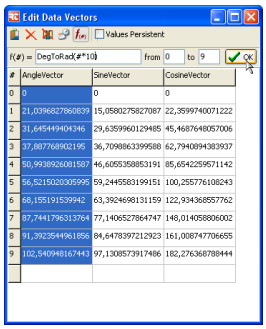
Outliers Support

On occasion, you may find that you have a large amount of correct data, but a number of data points do not fit to the others as a result of electrical noise (spikes), instrument failure, or other causes; this is often the case with time series. To assist you in cleaning out these values, you can mark them as outliers. Note however that the series components have

properties which decide whether data is shown with or without outliers. To mark a value as an outlier you can use the ***Outliers*** indexed property or the AddOutlier method.

Design time editing

During design time, vectors assigned to series will have incremental, random numbers as values. This will assist you in adjusting the colors and style for series visually. If you want to change this data, you can use a special component editor for this purpose. The editor will open when you double-click any **TRtDoubleVector** component or when you choose Edit from its context menu.



The edit window will display a table of all **TRtDoubleVector** components available on the form that are not linked to a database. The values inside the table can be edited by double-clicking or by pressing F2 within the active cell. Using the toolbar buttons you can delete the current selection, delete the complete column, display the column as a date/time value or open the function calculator line. The "Values Persistent" check box indicates whether all the values will be persisted – meaning that the values from the table will be loaded when the compiled program starts. Having starting values defined in this way may be useful for demo purposes.

The function calculator line can be used to recalculate the current column via an arbitrary formula. It can contain several operators compatible with Delphi or C++ syntax, plus the power operator, ^. Case is not significant when matching function names or the 'E' character used in scientific notation. Spaces and tabs are ignored.

The grammar follows the normal rules of arithmetic precedence, with ^ highest, * and / in the middle, and + and – lowest. Thus 1+2*3^4 is equivalent to 1+(2*(3^4)). Note that the power operator X^y is right-associative: 2^0.5^2 is thus equivalent to 2^(0.5^2). All other arithmetic operators are left-associative: 1-2-3 is equivalent to (1-2)-3. Parentheses can be used to force non-default grouping. Trigonometric functions such as sin(), cos(), etc., have the radiant as the argument, while the reciprocal angle functions have the radiant as the result.

Table: *Syntax for Calculation*

Variable	
#	The row index
Operators	
+	Plus
-	Minus
*	Multiply by
/	Divide by
^	To the power of
Functions	
abs()	Absolute value
arccos()	Inverse cosine
arccosh()	Inverse hyperbolic cosine
arcsin()	Inverse sine
arcsinh()	Inverse hyperbolic sine
arctan()	Arctangent
arctanh()	Inverse hyperbolic tangent
ceil()	Lowest integer greater than or equal to argument. The absolute value of the argument must be less than 2147483647.
cos()	Cosine
cosh()	Hyperbolic cosine
cotan()	Cotangent
DegToRad()	Degrees to radiant (same as: *pi/180)

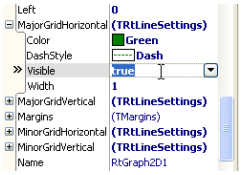
exp()	Exponential (power to base e)
floor()	Highest integer less than or equal to argument. The absolute value of the argument must be less than 2147483647.
frac()	Fractional part
int()	Integer part
ln()	Natural logarithm
lg(), log10()	Logarithm with base 10
log2()	Logarithm with base 2
RadToDeg()	Radiant to degrees
round()	Argument rounded to the nearest whole number.
sin()	Sine
sinh()	Hyperbolic sine
sqr()	Square
sqrt()	Square root
tan()	Tangent
tanh()	Hyperbolic tangent

Constants

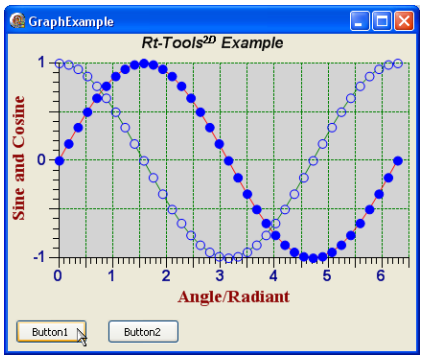
pi	3.14159…
e	2.71828...

2.2.3 The Cartesian Graph Control

The **TRtGraph2D** control is the container for all visual components that give the Cartesian plot as axes, series legends, etc..

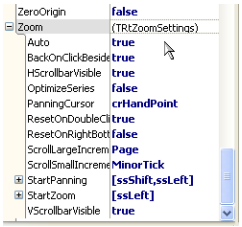


If you drop a **TRtGraph2D** component on a form, it will automatically create a set of primary axes at the left and the bottom, as described in the above example. These primary axes will be the reference of most of the data series. They are also the sources for the grid which can be activated using the relevant ***MajorGrid…*** properties.

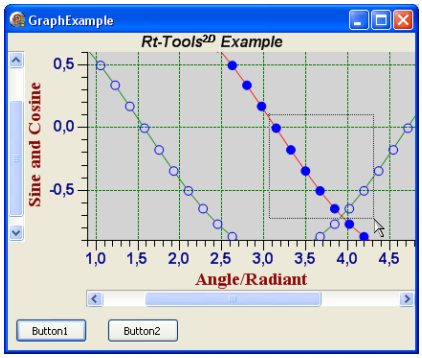


The graph can display a title centered at the top of the graph. The title text can be entered with the ***Caption*** property and made visible using the ***CaptionVisible*** property. In addition, you can define a ***DataAreaColor*** that differs from the ***BackColor*** of the graph, improving its presentation.

The background and data area can also be set up to be drawn as gradient.

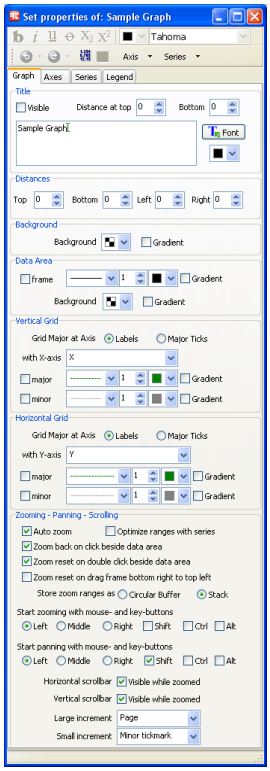


Your graph can also support auto-zooming and panning, by setting the relevant ***Zoom*** properties. You can show scrollbars appropriate to the zoomed range. Mouse gestures can be set that zoom back one level or reset the scaling when clicking or double-clicking beside the drawing area.



The various possibilities for zooming and panning are demonstrated in the "ZoomingAndPanning" example.

Design time editing

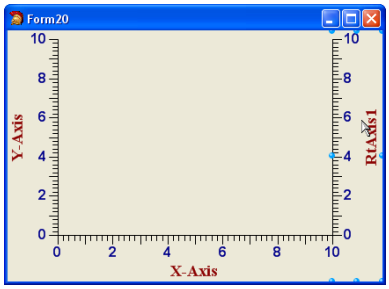


Since it may prove difficult to remember all of the numerous property names and appropriate settings, a component editor is available to help you tailor the layout of the graph to your needs. This editor opens when you double-click the **TRtGraph2D** control or choose *Edit* from its context menu.

2.2.4 The Axis Control

The **TRtAxis** control will normally be contained inside a **TRtGraph2D** control as its primary axis, as described above. Nevertheless, it can stand on its own – when demonstrating the settings, for example.

The main reason for supplying **TRtAxis** as a separate component and not embedding it exclusively within **TRtGraph2D** was the fact that it is preferable to be able to add an unlimited number of axes to each position of the graph using the mouse. This means that the graph can contain as many secondary axes as are needed.



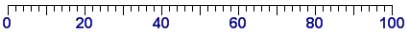
To add an additional axis to a graph, just select the **TRtAxis** component in the Rt-Tools2D Graph palette of the IDE and click inside the graph at the position (top, bottom, left or right) where the new axis should be placed.

You can select the new axis with a mouse click. The ***Position*** property can be modified to point to any place in the graph: the graph will update, automatically rearranging the bottom axis as it does so.

Another possibility is to use the secondary axis as a slave of the primary axis. In so doing, you can thus set the ***MasterAxis*** property to the primary axis which will supply the synchronized ranges. The ***IsSlaveAxis*** property will be automatically switched to *true*. Once done, the two axes remain synchronized.

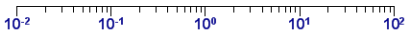
Scaling

The axis ***Scaling*** can be set using the relevant property. Six different settings are available:

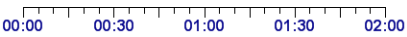


If *Normal* is set, the scaling will show equidistant ticks spaced linearly at positions related to the decimal system. E.g. 10, 20, 30...

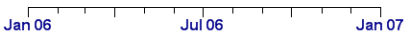
If *Log* is set, the scaling will show ticks spaced logarithmically and will show scale labels using decimal points or large numbers.

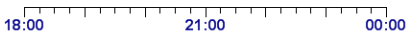


If *LogEE* is set, the ticks will be spaced logarithmically as above, but the scale labels will show numbers in exponential format.



The *Seconds* scaling is often used to display time series. The scaling will show equidistant ticks spaced linearly at positions related to the sexagesimal system, meaning that ticks are at positions as on a stop watch: at 5, 15, 30, 60 seconds, minutes and 6, 12, 24 hours.





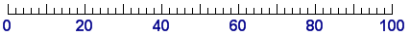
For data acquired by date and time, the *DateTime* setting is chosen. Depending on the range the axis shows years, months and days, or hours, minutes and seconds.

The *Rubrics* setting can be used to show data related to classes. The classes names should be the captions of the ***Rubrics***.

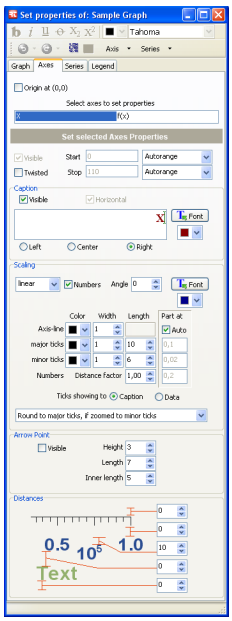
The text angle can be specified with the ***RubricsAngle*** property. You can link the rubrics to a database using the ***DataSource*** and ***RubricsField*** properties. This is demonstrated in the "AxisRubricsFromDatabase" example found in the "DatabaseAccess" directory.

Other Properties

The ***Twisted*** property can be used to generate an axis with an unusual direction. If set to *true*, horizontal axes will have low values at the right, vertical axes will have low values at the top. This setting makes sense when plotting infrared spectra using wave numbers as units, for example.



The ***TicksPointToData*** property controls the direction of the tick marks in relation to the graph series drawing area.



The other properties are explained in detail in the reference 138 section. Since it may prove difficult to remember all the numerous property names and appropriate settings a component editor is provided to help you tailor the layout of the axis to your needs. Open this editor by double-clicking the **TRtAxis** control or by choosing *Edit* from its context menu. Note that you can use the axes list box at the top to select more than one axis by using standard multiple selection methods such as Ctrl clicking with the mouse. The changes made will then apply to all the selected axes.

2.2.5 Series Components

The series components are used to hold information about the style and colors to be drawn in the graph. References to the axes are used to scale the data points as well as the vectors that hold the data point values. As shown in the above example, this can be done by setting the ***XAxis*** and ***YAxis*** properties for the scaling and ***XData*** and ***YData*** for the vectors. As the example also shows, the data vectors can share data, meaning that two different series can share the same X-data. You can even omit one of the ***XData*** or ***YData*** vectors: default index counter values will be substituted for the omitted property. In this case, the object inspector will show (Index).

Each vector assignment property of a series has corresponding ***...ValueTransformation*** events available. Specifying such a function will thus enable you to transform values from the source vector values into any other values you require. The "DataAcquisition" example program demonstrates one possible usage. Here, the original measurement was performed using a data acquisition card and a sample rate of 12.5 points/s. To display the correct experiment time , the X-index is converted by using the following formula:

Delphi:

```
function TAcquisitionMainForm.IntensityLineXValueTransformation(Sender: TObject; e:
TRtTransformEventArgs): Double;
begin
  Result := (e.Index+1)/12.5;   // convert data number to time in sec.
end;
```

C++

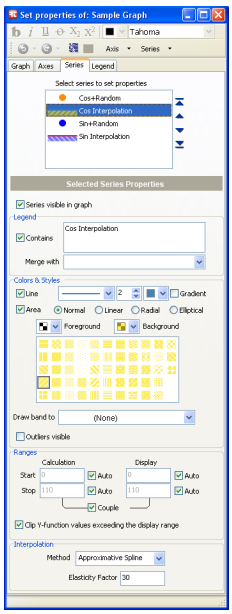
```
double __fastcall TAcquisitionMainForm::intensityLineXValueTransformation(TObject *Sender,
TRtTransformEventArgs &e)
{
   return (e.Index+1)/12.5;    // convert data number to time in sec.
}
```

If you place a series component on a form containing a **TRtGraph2D** component, the primary

axes of this graph will be automatically assigned as the ***XAxis*** and ***YAxis*** properties of the series.If you place the series first and the graph component second, you must then select the graph axes for the series. If you want to show data related to a secondary axis – such as a differential curve with the axis at the right – then you can simply place a right axis on the graph and assign it to the ***YAxis*** property.

The ***AutoUpdate*** property is normally set to true. This means that the graph will automatically update the series if new values have been added or if values have been changed. If you want to have more control about calculation – as might be the case with large data amounts, for example – it can be set to false. You can use the ***Calculate*** method to update the ranges and perform recalculation.

As with the data vectors 21, the series can be set to display or omit data points marked as outliers by using the ***OutliersVisible*** property.
The ***Caption*** property is used for the description of the series and is shown with the items in the **TRtLegend** 35 control. The caption can use extended formatting options 10. It can be included in the legend by setting the ***IncludeInLegend*** property to *true*. If ***IncludeInLegend*** property is *false* different series can merge their displayed items utilizing the ***MergeLegendItemWith*** property giving the parent series used to show the caption and own display items.



Most other settings for series components are specific to the series type. They set colors, styles, fills, etc.. Once assigned to a graph, these properties can be easily set using a component editor. The editor opens when you double-click the **TRtSeries** or when you choose *Edit* from its context menu. If you have more than one series set on the graph then these are listed in a box at the top. Once you select a series, you can alter its properties below. Series are normally drawn on the graph in the order of their creation, meaning that the last series created is drawn on top. However, you can fine-tune the layering of the different series by modifying the order in the top list: to do this, either drag the items using the mouse or use the arrow buttons provided.

The **TRtLineSeries** component is the simplest series component, showing poly lines, plus optional filled areas below.

This series type can be used if the single data points do not need to be shown separately. The line style is set via the ***Line.Color***, ***Line.DashStyle*** and ***Line.Width*** properties. The line display can be controlled using the ***Line.Visible*** property. The area below can be filled using the ***Area.BackColor*** and ***Area.ForeColor***, utilizing the ***Area.AreaStyle*** hatch fill. The display of the area is controlled by the ***Area.Visible*** property.

If you want to draw a band between two series you can specify a ***DrawBandTo*** series. If specified a filled area will be dawn between the current series and the other series specified by ***DrawBandTo***.

The **TRtPointSeries** component acts in the same way as TRtLineSeries 27 component above, but adds point symbols at each data point position.

The point symbols style is set via the ***PointSymbol.BorderColor***, ***PointSymbol.FillColor***, ***PontSymbol.BorderLineWidth***, ***PointSymbol.Size*** and ***PointSymbol.PointSymbol*** properties.

The **TRtPointWithErrorSeries** acts in the same way as the above TRtPointSeries 28, but adds error indicators to the point symbols.

The data points display the error indicators for each point and each direction individually, sized in accordance with the values specified in data vectors. If the vectors are set to zero, the relevant error indicator for the direction is not drawn. In this case, use the ***dXMinus***, ***dXPlus***, ***dYMinus*** and ***dYPlus*** properties to assign the relevant data. The "ErrorIndicators" example program gives you some hints on how they can be used with relative and absolute errors calculated by transform functions.

The **TRtBars** component is used to draw bar charts.

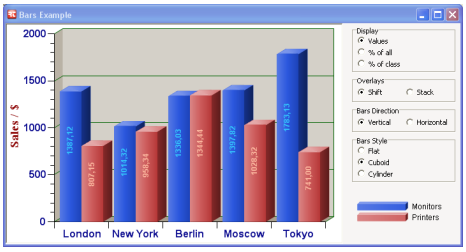
This series does not show any point symbols and lines to draw a curve, but uses bars pointing to the Zero-Line, where size represents the data points.

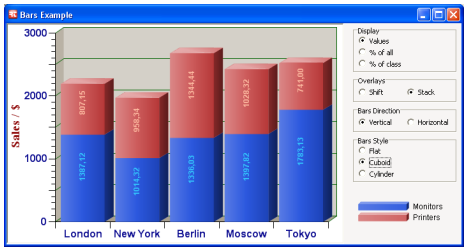
> The ***XData*** for vertical bars (and the ***YData*** for horizontal bars) holds the numerical data defining the start of each bar in X(Y)Data[i] and the end in X(Y)Data[i+1]. This means that the width of the bars must not be equally spaced but should be in ascending order. Note also that you have to add an extra X(Y) data value for the last data point if you want to show all points. If you do not assign any vector, then the index is used instead – which will meet this requirement automatically.

Please have a look at the "Bars" example project which explains most of the properties mentioned.

The bars can be displayed as simple flat rectangles, cuboids or cylinders, as specified via the ***BarStyle*** property. The bars are normally drawn vertically, but setting ***BarsHorizontal*** will draw the bars horizontally. The distance between the classes can be expressed relative to the bar classes width in % as ***ClassesDistance***. The 3D depth of the cuboids or cylinders is defined with the ***Depth3D*** property.

If the graph contains more than one bar series, they can be overlaid in two different ways. Setting the ***OverlayStyle*** to Shift will draw the bars side by side, sharing the class width. Setting ***OverlayStyle*** to Stack will draw the bars on top of each other, using the whole class width. The bars can contain labels showing individual values or any other caption. The position relative to the bar length is defined with the ***LabelPosition*** property.





If the ***LabelsFrom*** property is set to *Value*, the labels will display the single values of the data points using the format specified with the ***LabelFormat*** property. The format used is the same as the format for the ***FormatFloat*** function: a value of "1.234" with a format of "0.00" will thus display "1.23". An empty format will cause all significant digits to be displayed.

If the ***LabelsFrom*** property is set to *CaptionsList*, the label text is taken from the ***Captions*** string list. You can link the captions to a database using the ***DataSource*** and ***CaptionsField*** properties. This is demonstrated in the "BarLabelsFromDatabase" example found in the "DatabaseAccess" directory.

The **TRtBubbles** series component shows bubbles (circles or ellipses), whereby the radius represents a certain range.

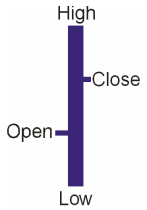
The ***RadiusData*** property must be assigned a data vector defining the radii of the bubbles. If the ***Elliptical*** property is set to false, the data points are displayed as (filled) circles; if not, ellipses are drawn giving a correct representation of the X-dimension as a radius in relation to the x-axis scale.

The **TRtArrows** series component can be used to display 2D vector fields as arrows.

The X and Y-component of the arrow direction and length are given with the ***DeltaX*** and ***DeltaY*** properties.

The **TRtOHLC** series component is used for financial charts, displaying the opening, highest, lowest and closing prices.

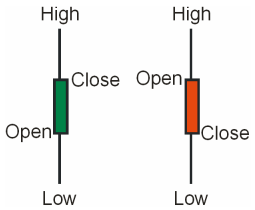
The bar size shows the highest price-lowest price range, the tick at the left shows the opening price and the tick at the right the closing price. The data vectors shown in this chart are supplied by the ***HighData***, ***LowData***, ***OpenData*** and ***CloseData*** properties.



The usage of this component is demonstrated in the "FinancialSeries" example project found in the "Financial" directory.

The **TRtCandleSticks** series component is used for financial charts, displaying opening, highest, lowest and closing prices.

The candle shows the highest price-lowest price range with the shadows, while the bar size shows the opening price and closing price range. Rising share prices will be indicated by the ***BullColor*** (green), while falling share prices are displayed using the ***BearColor*** (red). The data vectors shown in this chart are supplied by the ***HighData***, ***LowData***, ***OpenData*** and ***CloseData*** properties.



The usage of this component is demonstrated in the "FinancialSeries" example project found in the "Financial" directory.

2.2.6 Function Series Components

The following chapters describe series components that do not display the data points assigned: they perform special calculations based on the source data and display a result line.

As with the normal series components, source data is specified using the ***XData*** and ***YData*** vectors. In addition, you can supply a ***Weight*** vector. The data stored in this vector determines the statistical weights used in the calculation algorithm, yielding the result line of the function. The weight data points are understood to mean: "Take this data point Weight[i] times for the calculation result". If no ***Weight*** vector is assigned, all data points will be considered to be equal ($w[i]=1$).

The calculation algorithms can be applied to a specific range of X-values. To adjust this range to all the data available, set the ***AutoStart*** and ***AutoStop*** properties to *true*. If set to *false*, the range can be defined with the ***CalculationStart*** and ***CalculationStop*** properties.

Usually, you will want to display the calculated line in the X range you have chosen for

calculating the line. In this case, you should set the ***CoupleCalculationAndDisplay*** property to true. In cases where you want to show another display range than the one used for your calculation – when showing extrapolated values, for example – you can set this property to false and then set the ***DisplayStart*** and ***DisplayStop*** properties to the desired range. If the ***DisplayAutoStart*** and ***DisplayAutoStop*** properties are set, the display range will be set to the whole range of values of the source X-data.

In the "Events" tab of the object inspector you will find the ***OnCalculated*** event. This event is triggered every time the line is recalculated, meaning that you can use it to update labels showing calculated results, for example. The ***OnError*** event is triggered on every calculation error event raised by the relevant component. This can be used to enable custom error management, such as showing the error message in the status bar of the form, for example. If left unassigned, the errors will be raised normally – displaying a conventional error message box.

The **TRtLinearRegression** component is used to calculate straight line regressions.

Many processes in science can be expressed by straight line equations of the form $Y=a+bX$. Drawing straight lines passing through experimental data points is therefore a relatively common task. The slope and intercept of these lines are often used to calculate physical constants. The **TRtLinearRegression** calculates the regression line from the related source data using the method of least squares.

In some cases, it is preferable to calculate the line using a known intercept a. This is the case when a theory states that the line must cross zero, for example. To fix the offset a, you can set the ***FixedOffset*** property to *true* and assign the value of a to the ***Offset*** property.

After calculation, the results can be accessed using the properties:
Slope (b) and ***Offset*** (a) coefficients of the line equation
DeltaSlope and ***DeltaOffset*** their statistical errors
AverageX and ***AverageY*** means
VarianceX and ***VarianceY*** variances
Correlation coefficient (R) ***Probability*** (r^2),
and the sum of residuals ***SumOfResiduals***.

An example project "LinearRegression" demonstrates the usage.

> The component also calculates the linear regression if one or both axes are set to logarithmic scaling. The regression formula will be altered to $Y=a+b(lg(X))$; $lg(Y)=a+bX$ and $lg(Y)=a+b(lg(X))$ accordingly.

The **TRtGeneralLinearLeastSquares** component calculates a general linear least squares function line through the source points.

In some cases, you need to fit a model to your data points which can be expressed by functions of X which have linearly dependant coefficients and can be expressed with a function system as $Y = \sum_i a_i f_i(X)$ where the a_i are the linear coefficients of the best fit. In order to be able to calculate the optimization you must supply a function expressing the f_i by the ***BasisFunction*** property. Here you can specify any function dependant with X. The following Polynomial line for example is a special case of the General Linear Least Squares calculation the basis function for the polynomial is:

Delphi:

```
function TExampleForm.RtGeneralLinearLeastSquares1BasisFunction(Sender: TObject;
  e: TRtGeneralLinearFitingArgs): Double;
begin
  Result := IntPower(e.X,e.Index);
end;
```

As you see the basis function returns a function value dependant on the independent variable X and the column index of the solution matrix.

The ***Order*** of the function system to be calculated can be set using the relevant property.

After calculation the results can be accessed using the properties:
***Coefficients*[0]**, ***Coefficients*[1]** and ***Coefficients*[2]** the parameters a, b,c…
***DeltaCoefficients*[0]**, ***DeltaCoefficients*[1]** and ***DeltaCoefficients*[2]** standard deviation of the parameters a, b,c…
AverageX and ***AverageY*** means
VarianceX and ***VarianceY*** variances
Probability (r^2), and the sum of residuals ***SumOfResiduals***.

Please have a look at the "GeneralLinearLeastSquares" example project.

The **TRtPolynomial** component calculates a least squares polynomial line through the source points.

In some cases, it is better to apply a regression calculation giving the least squares fit for a polynomial $Y = a+bX+cX^2+....$ This component calculates this polynomial regression line. The ***Order*** of the polynomial to be calculated can be set using the relevant property.

After calculation the results can be accessed using the properties:
***Coefficients*[0]**, ***Coefficients*[1]** and ***Coefficients*[2]** the parameters a, b,c…
***DeltaCoefficients*[0]**, ***DeltaCoefficients*[1]** and ***DeltaCoefficients*[2]** standard deviation of the parameters a, b,c…
AverageX and ***AverageY*** means
VarianceX and ***VarianceY*** variances
Probability (r^2), and the sum of residuals ***SumOfResiduals***.

Please have a look at the "PolynomialRegression" example project.

The **TRtFittedLine** component is used for non-linear iterative (simplex) fitting

In many cases, the formula representing the physical law may be a non-linear function. In such cases, the calculations cannot solve a linear equation system. An iteration algorithm is thus applied, varying the parameters of the function to fit a minimum of variations from the data points. This component uses the Simplex algorithm to perform the non-linear fit.

As the function to be fitted can be an arbitrary function, you must supply a ***FittingFunction*** using the Event tab of the object inspector or enter a valid ***Expression*** property. This function can use as many variables as needed, specified with the ***Order*** property. Please note, that you must use the independent variable X at least once. The variable parameters must start with ***Coefficients*[0]** continuing up to the last parameter needed.

If no ***FittingFunction*** has been applied, the ***Expression*** property string is parsed during runtime for a valid function; if valid, this is used instead for the calculation. The expression can contain several operators compatible with Delphi or C++ syntax, plus the addition of the power operator, ^. Case is not significant when matching function names or the 'E' character used in scientific notation. Spaces and tabs are ignored.
The grammar follows the normal rules of arithmetic precedence, with ^ highest, * and / in the middle, and + and – lowest. Thus 1+2*3^4 is equivalent to 1+(2*(3^4)). Note that the power operator X^y is right-associative: 2^0.5^2 is equivalent to 2^(0.5^2). All other arithmetic operators are left-associative: 1-2-3 is equivalent to (1-2)-3. Parentheses can be used to force non-default grouping. Trigonometric functions such as sin(), cos(), etc., have the radiant as the argument, while the reciprocal angle functions have the radiant as the result.

Table: *Syntax for Expressions*

Variables	
a, b,..p	Coefficients varied during the iterative fit, at least a must appear
X	The independent variable, must appear once
Operators	
+	Plus
-	Minus
*	Multiply by
/	Divide by
^	To the power of
Functions	
abs()	Absolute value
arccos()	Inverse cosine
arccosh()	Inverse hyperbolic cosine
arcsin()	Inverse sine
arcsinh()	Inverse hyperbolic sine
arctan()	Arctangent
arctanh()	Inverse hyperbolic tangent
ceil()	Lowest integer greater than or equal to argument. The absolute value of the argument must be less than 2147483647.
cos()	Cosine
cosh()	Hyperbolic cosine
cotan()	Cotangent
DegToRad()	Degrees to radiant (same as: *pi/180)
exp()	Exponential (power to base e)
floor()	Highest integer less than or equal to argument. The absolute value of the argument must be less than 2147483647.
frac()	Fractional part
int()	Integer part
ln()	Natural logarithm
lg(), log10()	Logarithm with base 10
log2()	Logarithm with base 2
RadToDeg()	Radiant to degrees
round()	Argument rounded to the nearest whole number.
sin()	Sine
sinh()	Hyperbolic sine
sqr()	Square
sqrt()	Square root
tan()	Tangent
tanh()	Hyperbolic tangent
Constants	
pi	3.14159…

The quality of the fit is strongly dependent on the starting values for the ***Coefficients***. Because the algorithm can only find the local minima of the optimization function, these settings must be appropriate for being located in the region of the global minimum.

With the ***DoCovariance*** property you can specify whether you want to optimize the fit for a minimum variance (false) or covariance (true). In most cases you will want to select the variance option, because this is the classical least squares approach. The theory states that the deviations in the formula result are not caused by uncertainties in the independent variable X. The optimization thus minimizes the sum of squares of distance of each resulting formula Y value to the experimental Y value. In some cases, especially with functions with steep and flat parts, the deviations in the steep part would always dominate the optimization. Thus you could consider applying a weighting function to compensate these effects. This

component gives you another opportunity for solving this problem using the covariance as an optimization criterion instead. As the covariance is the product of the deviations in the X- and Y-directions of each point related to the calculated formula, the sum of the absolute values of covariance can give a better optimum. This approach would not overestimate any part of the function, resulting in visibly better fits. However, the covariance should only be applied for the special functions mentioned above. You also must take care to ensure the fitting function is monotone, because the calculation of the covariance needs a single X and Y function point related to one experimental point.

After calculation, the results can be accessed using the properties:
***Coefficients*[0]**, ***Coefficients*[1]** and ***Coefficients*[2]** the parameters a, b,c...
AverageX and ***AverageY*** means
VarianceX and ***VarianceY*** variances
Probability (r^2), and the sum of residuals ***SumOfResiduals***.

The "NonLinearRegression" example project demonstrates the usage of this component.

The **TRtMovingAverage** series component is used to show smoothed lines calculated by a moving average.

The usage of this component is demonstrated in the "FinancialSeries" example project found in the "Financial" directory.

This method is especially helpful for display trend lines in financial charts. The values of the source data are summed up over a specified range and divided by the value count, resulting in a smoothed line. For financial charts, historical price data is usually taken for the sum. This corresponds to ***AveragingDirection*** set to *Downwards*. For other applications it might be useful to sum *UpAndDownwards*. This means the smoothed line follows the data points and the trend is not shifted to the past.

The component can be set up to use different summing algorithms via the ***AveragingMethod*** :

Simple will just use the sum and divide by the point count

$$\overline{Y}_i = \frac{\sum_{k=i-n}^{n} Y_k}{n}$$

LinearWeighted will weight the points linearly in relation to the distance to the point of interest

$$\overline{Y}_i = \frac{\sum_{k=i-n}^{n} Y_k \left(1 - \left|X_k - X_i\right|\right)/n}{n}$$

ExponentiallyWeighted will weight the points exponentially in relation to the distance to the point of interest

$$\overline{Y}_i = \frac{\sum_{k=i-n}^{n} Y_k \left(\exp\left(u\,v\left|X_k - X_i\right|\right)\right)}{n} \qquad u = \ln\left(ExponentialFactor\right) \qquad v = \frac{n}{X_i - X_n}$$

Normally the summing range is given with the count of values to be summed. This behavior is set via the property ***AveragingRangeMethod*** *NasCount*. In some cases, especially where the X-values are not equally spaced, you can select *NasXDistance*. In this case, the averaging range is taken, comparing the X-distance of the values to the point of interest ***N***.

The **TRtInterpolation** component will draw an interpolated line between the source data points.

The component uses a third degree polynomial between every source data point to calculate the interpolated points. There are numerous algorithms available for deriving the coefficients for these polynomials. Via the ***InterpolationMethod*** property you can select one of the following:

Akima can interpolate curves exactly passing the source data points but without a steady second derivative. Stepped curves can thus be interpolated without applying stiffness factors. The result will have no swingers.

CubicSpline will use the classical (normal) spline algorithm, exactly passing the source data points. The interpolated curves are steady and steady in the second derivative. In some cases this generates large swingers.

ApproximativeSpline calculates an optimized function between all source data points using a special weighting function, letting the source data points influence the linearly-optimized interpolating function. It can thus show pleasantly smooth interpolations, especially on noisy source data.

Please see the "AkimaInterpolation", "CubicSpline" and "ApproximativeSpline" examples in the "Interpolation" directory for the effects of the different methods.

As the two first interpolation methods *Akima* and *CubicSpline* generate interpolated lines passing the source data points, this can result in very noisy lines. A smoothing calculation can therefore be executed before interpolation, by setting the ***SmoothPoints*** property to *true.* The ***SmoothingRange*** property gives the range that is used to smooth the values. The smoothed values will be calculated according to a moving average weighted by the distance of the average points. The advantage of defining a range instead of a number of moving average points is that a range also gives a smooth function in parts where the density of data points varies a great deal. Smoothing is only achieved if the range is large enough to cover at least one different data point. The larger the distance, the more values will be included in the average, giving a smoother line.

The *ApproximativeSpline* does not need to smooth before interpolation because the interpolation algorithm itself produces smooth results using the appropriate ***ElasticityFactor***. As described above, a natural spline will have a steady second derivative. The approximative spline interpolation function uses weights for the data points to calculate the interpolating polynomial. The higher the weights, the more the interpolated function will be forced to pass the points – just as if the function were to be attracted by a magnet. High weights will force the spline to pass the point; low weights result in a stiffer interpolation in-between the points. The weights used to derive the interpolation polynomial are taken from the assigned ***Weight*** and multiplied with the ***ElasticityFactor***. If no weights have been assigned, the ***ElasticityFactor*** is used for every point.

Since the interpolating polynomials are worked out during calculation, this enables the local ***Minima*** and ***Maxima*** to be derived from the function and obtained with the relevant arrays.

The **TRtDifferential** component gives the differential of the interpolated line between the source data points.

As shown above, the different interpolation methods available will calculate an interpolating polynomial between the source data points. This polynomial can be analytically differentiated, resulting in the differential line.

The **TRtIntegral** component gives the integral of the interpolated line between the source data points.

As shown above, the different interpolation methods available will calculate an interpolating polynomial between the source data points. This polynomial can be analytically integrated, resulting in the integral line.

The **TRtDiscreteFourier** component gives the Discrete Fourier Transformation of the source data points.

It shows the frequency factors with their amplitudes. The phases of the complex factors can be displayed enabling the embedded ***Phases*** series. It works best if the source data count is a power of 2. In that case it can internally calculate using FFT. The resulting frequency spectrum however will only contain factors up the half count to avoid the Alias-Effect.

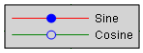
See the "Fourier" example project for usage.

2.2.7 The Legend

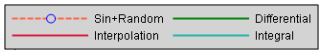
The **TRtLegend** control displays the styles and captions of the series contained in the graph.

When you place the control on a form containing a **TRtGraph2D** control, this control will be automatically assigned the ***Graph*** property of the legend. Newly-created controls will have the ***Position*** set to *FreeFloating*. This means that the user can drag the legend with the mouse to any desired position even during runtime. Setting the ***Position*** to other values will adjust it relative to the graph dimensions or position of the axes.

In order to adjust the layout to the drawing space available, the control provides the ***ItemsOrder*** property.



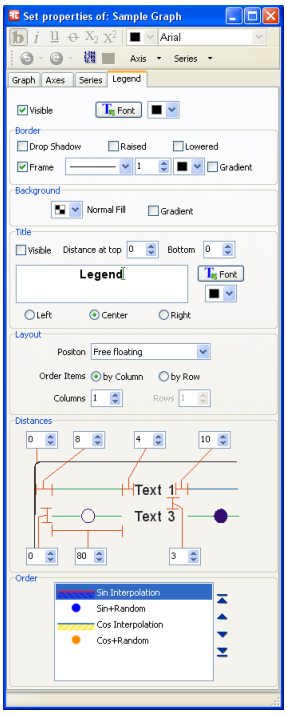
If set to *ByColumn*, the items in the list are organized vertically into a column.



If the legend does not fit the height of the graph, you might set ***Columns*** >1.

If the ***ItemsOrder*** is set to *ByRow*, the items in the list are organized horizontally into a row.

If the legend does not fit the width of the graph, you might set ***Rows*** >1.



The order of the items in the list defaults to the layer order of the series in the graph; this itself defaults to the creation order of the series components. If you want to alter the order in the list you can do this by modifying the ***ItemsOrderList*** using a special property editor.

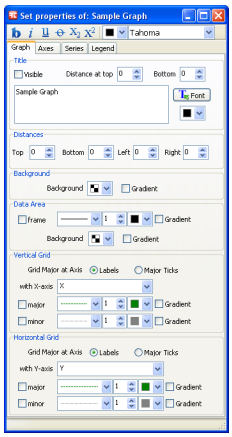
Since it might be difficult to remember all the numerous property names and the appropriate settings, a component editor is provided to help you tailor the layout of the legend to your needs. Open the editor by either double-clicking the **TRtLegend** control or by choosing Edit from its context menu.

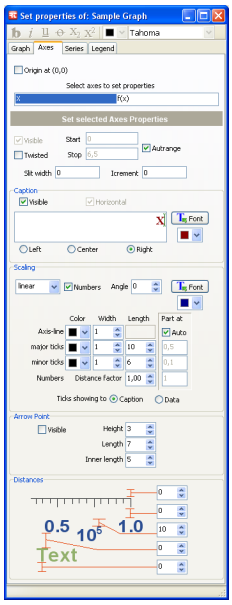
2.2.8 The Graph Settings Tool

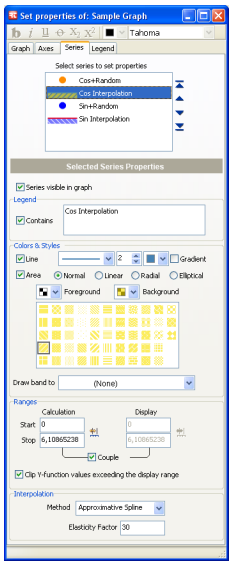
The **TRtGraphSettingsTool** provides a tool window for setting all possible options for the graph.

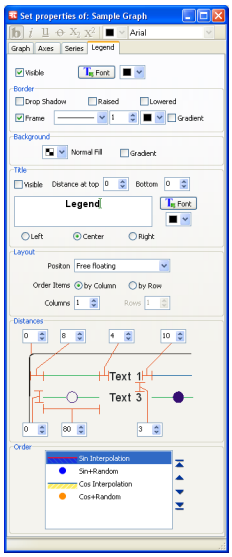
Since the user of your program will want to change the colors of lines, alter axes settings, change captions, etc., you need to provide your program with the functionality for setting all these properties. The easiest way to do this is to add a **TRtGraphSettingsTool** component to the form and call its ***Show*** method. This will open the tool window, displaying the settings for the assigned ***Graph***, the axes, the series and the legend in separate tabs.

Using the toolbar at the top, you can set the extended format options 10 if the focus is in a caption edit field. The various settings correspond to almost all the properties mentioned in the chapters above that describe the graph components. A short hint text explains each field.









2.2.9 Movable Markers

Occasionally you will need to mark something in a graph. You may want to mark a calculation range with vertical lines, show a level using a horizontal line, indicate a position using a crosshair or annotate peaks with an arrow and a caption. RtTools2D provides a set of components that give you all these possibilities.

Common to all these controls is that fact that the user can drag their objects around the graph with the mouse. The position of the object in the graph relates not to left/top points of the screen coordinates but to graph-specific ***X*** and ***Y*** values in relation to the ***XAxis*** and ***YAxis*** assigned, giving the position of the hot spot of the object in graph world coordinates. This has the advantage that the markers stay at the correct position even when zoomed.

If you set the ***SnapToSeriesValues*** property to *false*, the object can be moved around with the mouse without any restrictions. If set to *true* and the ***SnapToSeries*** property assigns a point series inside the graph, the movement will be limited to the series point positions. This means that the object hot spot will snap to the series point position.

In addition to this functionality a marker snapped to a series stays synchronized with the snapped point. This means, if the value of the series point changes, the position of the marker will change accordingly. If a corresponding value is deleted the marker will also be deleted.

The markers can have an optional ***Caption*** which can be shown on any end or in the middle defined with the ***CaptionPosition*** property.

If the ***ClipToData*** property is set to *true*, the display of the marker object will be limited to the drawing area of the graph points.

Please have a look at the "Markers" example project which demonstrates the components mentioned above.

> Transparent backgrounds are not supported with moving markers by Delphi 6. Thus please set the graph background color to any non transparent color. Users of Delphi 7 and higher versions however can also use transparency with movable markers


The **TRtVerticalMarker** component can be used to mark X-values.

By placing this control on a graph you can mark X-values with a vertical line.


The **TRtHorizontal** component can be used to mark Y levels.

By placing this control on a graph you can mark Y values with a horizontal line.


The **TRtCrossHair** component can be used to mark X/Y data points.

By placing this control on a graph you can mark X/Y data points with a horizontal and a vertical line, with the hot spot at the line intersection. The crosshair can have a ***Caption*** at the cross in any of the 4 quadrants specified by the ***CaptionPosition*** property.


The **TRtLabelWithArrowMarker** component can be used to mark X/Y data points with an arrow and a caption.

By placing this control on a graph you can mark X/Y data points with an object composed of an arrow pointing to the hot spot and a caption at its end. The arrow can point in any direction and can contain an optional bend, bringing it level with the caption.


If the ***HoverGrips*** property has been set to *true*, moving the mouse over the marker during runtime will display some hovering handles, showing active mouse positions.

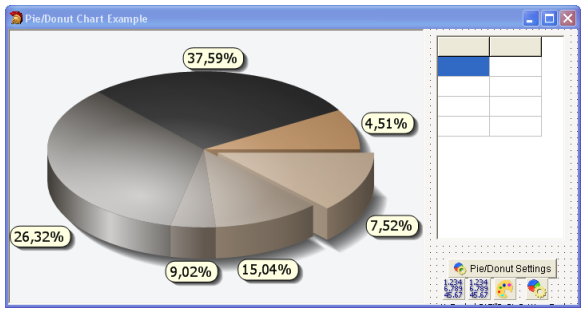
The mouse cursor will change at the end of the arrow inside the small bullet handle, indicating that here the user can drag the arrow end to any position, to give any angle or length required.

If the mouse cursor is inside the small frame grip, the mouse cursor will again change, indicating that the bent part of the arrow can be shortened or lengthened by dragging the mouse.

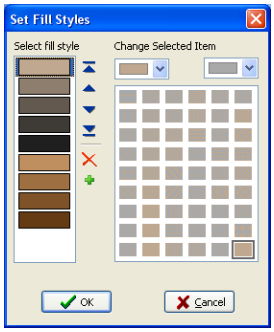
Moving the mouse to the text end handle changes the mouse cursor to indicate that the caption can be dragged relative to the arrow end.

2.2.10 Pie/Donut Charts

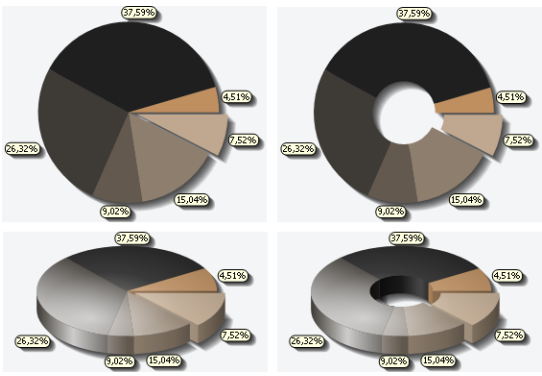
The **TRtPieChart** control was designed to display Pie and Donut charts. It also can be used as container for the relevant TRtPieLegend 39 control which displays the legend for the pie segment items.



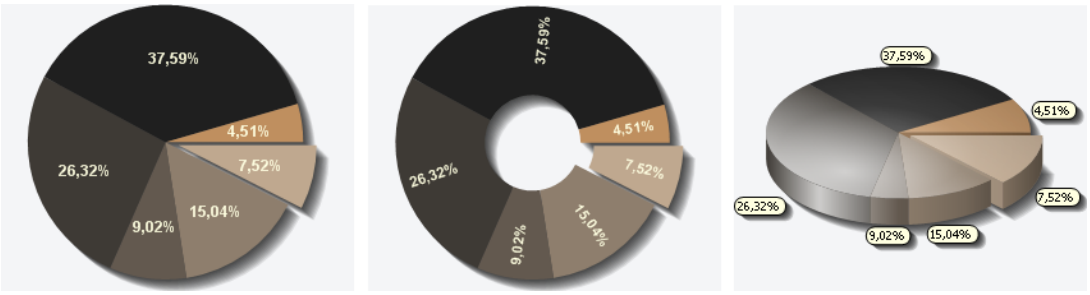
If you drop a **TRtPieChart** component on a form, it will automatically create a data vector for the ***Values*** to be displayed. Additionally you can supply data ***ExplodePercent*** to be interpreted as percentage of radial shift of the relevant pie segment. The fill settings of the pie segments can be altered with a **TRtFillPalette** component assigned to the ***SegmentSettingsPallete*** property which stores the fore and back colors and the fill styles for the segments in a list.



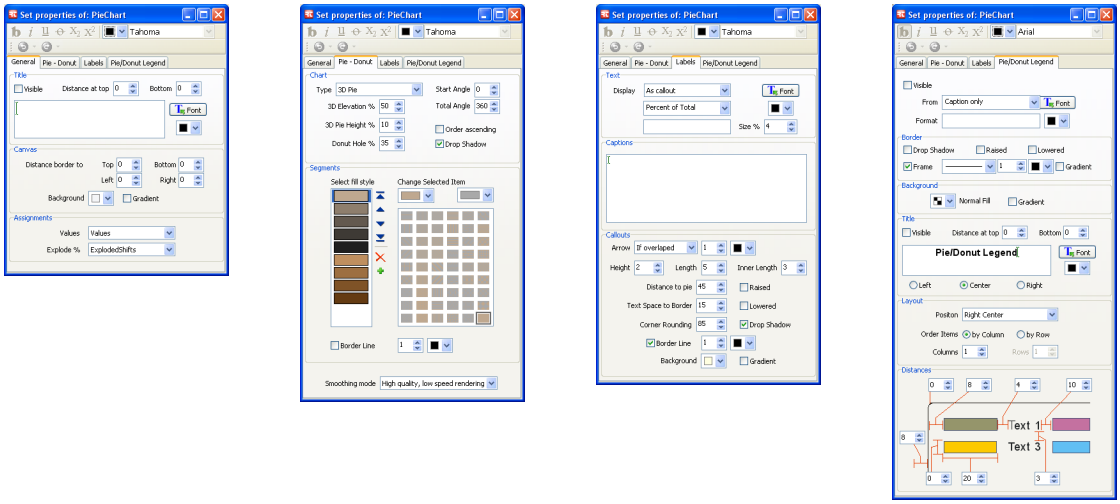
The fill settings can be altered within a special component editor double clicking the palette icon or double clicking the ***SegmentSettingsPallete*** property in the object inspector of the pie chart control. Within this editor you can add new items, delete existing items, reorder and change the individual settings.



The Pie Chart can be displayed as flat pie, flat donut 3D pie and 3D donut setting the ***PieChartType*** property.



The labels at the segments can be invisible or placed horizontally or angular inside the segments or as callouts setting the ***LabelPosition*** property. The content of the labels can be set to contain a textual caption, the values, the percentage or combinations of values and caption using the ***LabelFrom*** property.
Since it may prove difficult to remember all of the numerous property names and appropriate settings, a component editor is available to help you tailor the layout of the graph to your needs. This editor opens when you double-click the **TRtPieChart** control or choose *Edit* from its context menu.



The **TRtPieLegend** control displays the fill settings and relevant text for the pie segment items.

When you place the control on a form containing a **TRtPieChart** control, this control will be automatically assigned the ***PieChart*** property of the legend. If you place the legend to a **TRtPieChart** object the ***Position*** set to a position corresponding to the mouse cursor position.

The ***ItemsOrder, Columns, Rows*** and ***ItemsOrder*** properties have identical meanings as in the standard graphs TRtLegend 35.

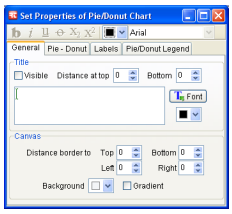
In addition the content of the labels can be set to contain a textual caption, the values, the percentage or combinations of values and caption using the ***LabelFrom*** property similar as with the pie segment labels.

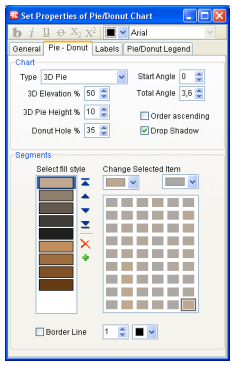
Since it might be difficult to remember all the numerous property names and the appropriate settings, a component editor is provided to help you tailor the layout of the legend to your needs. Open the editor by either double-clicking the **TRtPieLegend** control or by choosing *Edit* from its context menu (see above 39).

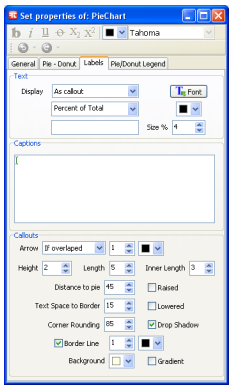
The **TRtPieSettingsTool** provides a tool window for setting all possible options for the pie/donut chart.

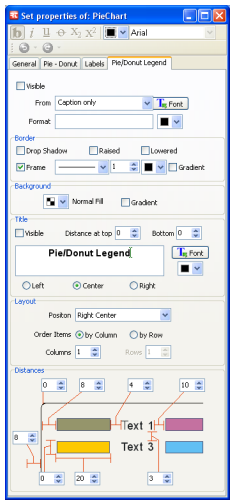
Since the user of your program will want to change the colors of the segments, borders, callout arrows, change captions, etc., you need to provide your program with the functionality for setting all these properties. The easiest way to do this is to add a **TRtPieSettingsTool** component to the form and call its ***Show*** method. This will open the tool window, displaying the settings for the assigned ***PieChart***, general, pie/donut, labels and the legend in separate tabs.

Using the toolbar at the top, you can set the extended format options 10 if the focus is in a caption edit field. The various settings correspond to almost all the properties of the **TRtPieChart**, **TRtFillPalette** and **TRtPieLegend** components. A short hint text explains each field.









2.3 Graph Setings Frames

The "Rt-Tools2D Frames" category of the tool palette of the IDE contains Frames that that have been developed to assist you in building up the user interface of your program. They are tailored to assist you in setting-up all the available properties in the Cartesian plot components, namely:

- TRtGradientFrame 183 - Used to setup gradients for lines, areas and fonts.
- TRtGraphSettingsFrame 183 - Used to setup general properties of the Cartesian graph. Equivalent to the first page 36 of the settings tool
- TRtAxisSettingsFrame 184 - Used to set the options of the selected axes. Equivalent to the lower part of the second page 36 of the settings tool.
- TRtSeriesSettingsFrame 185 - Used to set the properties of the selected series. Equivalent to the lower part of the third page 36 of the settings tool.
- TRtLegendSettingsFrame 186 - Used to setup the options of the legend. Equivalent to the fourth page 36 of the settings tool.
- TRtPieGeneralSettingsFrame 186 - Used to setup general properties of the pie chart. Equivalent to the first page 40 of the relevant settings tool.
- TRtFillPaletteFrame 186 - Used to setup the colors an fill styles of a fill palette. Used in the segments group of the second page 40 of the settings tool.
- TRtPieDonutFillFrame 187 - Used to setup the styles of the pie chart. Equivalent to the second page 40 of the settings tool.

- TRtPieLabelsFrame 187 - Used to setup the labels options for the pie chart. Equivalent to the third page 40 of the settings tool.
- TRtPieLegendSettingsFrame 187 - Used to setup the options for the legend at the pie chart. Equivalent to the fourth page 40 of the settings tool.
- TRtSelectSeriesFrame 188 - Used to select the visibility of series of a graph in a checked list box.

2.4 Redistributables

The following RtTools2D library files may be redistributed royalty free by customers who have purchased a valid license:

- RtTools2D_Delphi6_Runtime.bpl - for programs using runtime libraries with Borland Delphi 6.
- RtTools2D_Delphi7_Runtime.bpl - for programs using runtime libraries with Borland Delphi 7.
- RtTools2D_Studio2005_Win32_Runtime.bpl - for programs using runtime libraries with Borland Delphi 2005 Win32 personality.
- RtTools2D_Studio2005_net_Runtime.dll - for programs using runtime libraries with Borland Delphi 2005 Vcl. net. personality.
- RtTools2D_Studio2006_Win32_Runtime.bpl - for programs using runtime libraries with Borland Developer Studio 2006 Delphi Win32 and C++ personalities.
- RtTools2D_Studio2006_net_Runtime.dll - for programs using runtime libraries with Borland Delphi 2006 Delphi Vcl. net. personality.
- RtTools2D_Studio2007_Win32_Runtime.bpl - for programs using runtime libraries with CodeGear RAD Studio 2007 Delphi Win32 and C++ personalities.
- RtTools2D_Studio2007_net_Runtime.dll - for programs using runtime libraries with CodeGear RAD 2007 Delphi Vcl. net. personality.
- RtTools2D_Studio2009_Win32_Runtime.bpl - for programs using runtime libraries with CodeGear RAD Studio 2009 Delphi Win32 and C++ personalities.
- RtTools2D_Studio2010_Win32_Runtime.bpl - for programs using runtime libraries with Embarcadero RAD Studio 2010 Delphi Win32 and C++ personalities.
- RtTools2D_StudioXP_Win32_Runtime.bpl - for programs using runtime libraries with Embarcadero RAD Studio XP Delphi Win32 and C++ personalities.
- RtTools2D_StudioXP2_Win32_Runtime.bpl - for programs using runtime libraries with Embarcadero RAD Studio XP2 Delphi Win32 and C++ personalities.

Part III

3 Reference

3.1 Supporting Units

3.1.1 TRtCalculation Class

RtTools2D supports various calculated line series components. All these components have the common task of calculating particular results using an algorithm working with X/Y source data in a defined X range. It may prove useful to employ these algorithms without displaying any line output to a graph. All calculations used are based on the **TRtCalculation** class:

Unit/Namespace: `RtFunctionSeries`

Declaration

Delphi: `TRtCalculation = class;`
C++: `class TRtCalculation : public System::TObject;`

Public Variables

Delphi: `XData, YData, Weight: TRtCustomDoubleVector`94`;`
C++:
```
Rtvectors::TRtCustomDoubleVector* XData;
Rtvectors::TRtCustomDoubleVector* YData;
Rtvectors::TRtCustomDoubleVector* Weight;
```
↳ The X,Y and weight data vectors used for calculation. If weight is set to nil the calculation uses equally-weighted data points.

Delphi: `Start, Stop: Double;`
C++: `double Start; double Stop;`
↳ The X-values range of the calculation. Start defaults to *-MaxDouble*, Stop defaults to *MaxDouble*. Set to limited range if necessary.

Delphi: `OutliersVisible: Boolean;`
C++: `bool OutliersVisible;`
↳ Set to *true* if values indicated as outliers should be considered for calculation.

Delphi: `mXValueTransformation, mYValueTransformation, mWeightValueTransformation: TRtValueTransformation`98`;`
C++:
```
Rtseries::TRtValueTransformation mXValueTransformation;
Rtseries::TRtValueTransformation mYValueTransformation;
Rtseries::TRtValueTransformation mWeightValueTransformation;
```
↳ Events triggered every time an X/Y/Weight value is needed for calculation. Leave as *nil* to leave data unmodified.

Public Methods

Delphi: `constructor Create;`
C++: `TRtCalculation(void);`
↳ Create the calculation class, set ***Start*** to *-MaxDouble*, ***Stop*** to *MaxDouble*.

Delphi: `function Xi(Idx: Integer): Double; virtual;`
`function Yi(Idx: Integer): Double; virtual`
`function Wi(Idx: Integer): Double;`
C++: `virtual double Xi(int Idx);`
`virtual double Yi(int Idx);`
`virtual double Wi(int Idx);`

↳ Get X,Y, Weight data with index (Idx) and transform function if specified (for internal use only).

Delphi: `IsValid(Idx: Integer): Boolean;`
C++: `bool IsValid(int Idx);`

↳ Returns if X/Y/Weight data point is valid. Is *true* if no outliers at index (Idx) and ***OutliersVisible*** is *true* (for internal use only).

Delphi: `procedure Calculate; virtual;`
C++: `virtual void Calculate(void);`

↳ Executes the calculation algorithm.

3.1.2 TRtLinearRegressionCalculation Class

This class is used to calculate straight line equations $Y=a+bX$ using the data vectors. It is used internally by the TRtLinearRegression 30 series line component. Nevertheless, it can be used separately to perform calculations without displaying any graph lines.

Unit/Namespace: `RtLinearRegression`

Declaration

Delphi: `TRtLinearRegressionCalculation = class(TRtCalculation);` 44
C++: `class TRtLinearRegressionCalculation : public Rtfunctionseries::TRtCalculation;`

Public Variables

Delphi: `Slope, DeltaSlope, Offset, DeltaOffset: Double;`
C++: `double Slope; double DeltaSlope; double Offset; double DeltaOffset;`

↳ The offset a and slope b of the regression line and their statistical errors. If ***FixedOffset*** is set to *true*, the ***Offset*** defines the intersection of the y-axis.

Delphi: `AverageX, AverageY, VarianceX, VarianceY, Probability, Correlation, SumOfResiduals: Double;`
C++: `double AverageX; double AverageY; double VarianceX; double VarianceY; double Probability; double Correlation; double SumOfResiduals;`

↳ Statistical results.

Delphi: `FixedOffset: Boolean;`
C++: `bool FixedOffset;`

↳ Set to *true* to force the line to intersect at the pre-defined ***Offset***.

Delphi: `XLog, YLog: Boolean;`
C++: `bool XLog; bool YLog;`

↳ Set to true if the coordinate should be treated as (10th) logarithmic.

XLog *false*, ***YLog*** *false* ⇨ $Y=a+bX$
XLog *true*, ***YLog*** *false* ⇨ $Y=a+b(lg(X))$
XLog *false*, ***YLog*** *true* ⇨ $lg(Y)=a+bX$
XLog *true*, ***YLog*** *true* ⇨ $lg(Y)=a+b(lg(X))$

Public Method

Delphi: `procedure Calculate; override;`
C++: `virtual void Calculate( void)`
↳ Calculate the straight line linear regression.

3.1.3 TRtGeneralLinearLeastSquaresCalculation Class

This class is used to calculate a general linear least squares optimized base function system $Y = \sum_i a_i f_i(X)$ using the data vectors. It is used internally by the TRtGeneralLinearLeast-Squares 124 series line component. Nevertheless, it can be used separately to perform calculations without displaying any graph lines.

Unit/Namespace: `RtGeneralLinearLeastSquares`

Declarations

Delphi:
```
TRtGeneralLinearFittingArgs = record
  X: Double;
  Index: Integer;
end;
```
C++:
```
struct TRtGeneralLinearFittingArgs
{
      public: double X;
      public: int Index;
};
```
↳ The record used to pass the basis function actual index and independent variable X to the fitting function supplied.

Delphi:
```
TRtGeneralLinearFittingHandler = function( Sender: TObject; e:
TRtGeneralLinearFittingArgs): Double of object;
```
C++:
```
typedef double __fastcall ( __closure
*TRtGeneralLinearFittingHandler)( System::TObject* Sender,
const TRtGeneralLinearFittingArgs &e);
```
↳ The definition of the fitting basis function.

Delphi:
```
TRtGeneralLinearLeastSquaresCalculation = class( TRtCalculation
44 ;
```
C++:
```
class TRtGeneralLinearLeastSquaresCalculation : public
Rtfunctionseries::TRtCalculation;
```

Public Variables

Delphi: `Coeffs:`
C++: `DynamicArray<double >  Coeffs;`
↳ The array returns the calculated coefficients a_i

Delphi: `Coeffs: TDoubleDynArray;`
C++: `DynamicArray<double >  Coeffs;`
↳ The array returns the calculated coefficients a_i

Delphi: `DeltaCoeffs: TDoubleDynArray;`
C++: `DynamicArray<double >  Coeffs;`
↳ The array returns the calculated standard deviations of the coefficients a_i

Delphi: `AverageX, AverageY, VarianceX, VarianceY, Probability, QRel, SumOfResiduals: Double;`
C++: `double AverageX; double AverageY; double VarianceX; double VarianceY; double Probability; double QRel; double SumOfResiduals;`
↳ Statistical results.

Public Property

Delphi: `property Order: Integer read FOrder write SetOrder;`
C++: `__property int Order = {read=FOrder, write=SetOrder, nodefault};`
↳ The order of the function system. Will also set the size of the ***Coeffs*** and ***DeltaCoeffs*** arrays correctly.

Events

Delphi: `property BasisFunction: TRtGeneralLinearFittingHandler read mGetBasisFn write mGetBasisFn;`
C++: `__property TRtGeneralLinearFittingHandler BasisFunction = {read=mGetBasisFn, write=mGetBasisFn};`
↳ Your user function that returns the fitting basis function values at a specified *Index* and *X* value..

Public Method

Delphi: `procedure Calculate; override;`
C++: `virtual void Calculate( void)`
↳ Calculate the general linear regression curve coefficients.

3.1.4 TRtPolynomialCalculation Class

This class is used to calculate a polynomial $Y = a+bX+cX^2+...$ using the data vectors. It is used internally by the TRtPolynomial 31 series line component. Nevertheless, it can be used separately to perform calculations without displaying any graph lines. It is identical with the general linear least squares calculation setting the ***BasisFunction*** to *IntPower(e.X,e.Index).*

Unit/Namespace: `RtPolynomial`

Declaration

Delphi: `TRtPolynomialCalculation = class(`
`TRtGeneralLinearLeastSquaresCalculation` 46 `;`
C++: `class TRtPolynomialCalculation : public TRtGeneralLinearLeastSquaresCalculation;`

3.1.5 TRtSimplexFit Class

This class is used to calculate regressions to functions $Y = f(X)$ for the non-linear behavior of their coefficients a, b... given the correct starting values. The iteration algorithm used is the Simplex optimization. This method is used internally by the TRtFittedLine 31 series line component. Nevertheless, it can be used separately to perform calculations without displaying any graph lines.

Unit/Namespace: `RtFitting`

Declarations

Delphi:
```
TRtFittingFunctionArgs = record
  X: Double;
  Coefficients: TDoubleDynArray;
end;
```
C++:
```
struct TRtFittingFunctionArgs
{
      public: double X;
      DynamicArray<double >  Coefficients;
};
```
The record used to pass the corner point coefficients and independent variable X to the fitting function supplied.

Delphi:
```
TRtFittingFunctionHandler = function(Sender: TObject; e:
TRtFittingFunctionArgs): Double of object;
```
C++:
```
typedef double __fastcall (__closure
*TRtFittingFunctionHandler)(System::TObject* Sender, const
TRtFittingFunctionArgs &e);
```
The definition of the fitting function.

Delphi:
```
TRtFittingCornerHandler = procedure(Sender: TObject;
Coefficients: TDoubleDynArray) of object;
```
C++:
```
typedef void __fastcall (__closure *TRtFittingCornerHandler)
(System::TObject* Sender, TDoubleDynArray Coefficients);
```
The definition of the fitting corner method.

Delphi:
```
TRtSimplexFit = class(TRtCalculation);
```
44

C++:
```
class TRtSimplexFit : public Rtfunctionseries::
TRtCalculation;
```

Public Properties

Delphi:
```
property Iterations: Integer read FIterations write
FIterations;
```
C++:
```
__property int Iterations = {read=FIterations,
write=FIterations, nodefault};
```
The number of iterations used for the optimization of the non-linear function.

Delphi:
```
property Expression: string read FExpression write
SetExpression;
```
C++:
```
__property AnsiString Expression = {read=FExpression,
write=SetExpression};
```
Fitted expression as string. Only parsed if the ***FittingFunction*** is not defined.

Delphi:
```
property Coefficients: TDoubleDynArray read FCurrentCorner
write SetCoefs;
```
C++:
```
__property TDoubleDynArray Coefficients=
{read=FCurrentCorner, write=SetCoefs};
```
The array of coefficients fitted. Please supply correct starting values.

Delphi: `property DoCovariance: Boolean read FDoCovariance write FDoCovariance;`
C++: `__property bool DoCovariance = {read=FDoCovariance, write=FDoCovariance, nodefault};`
- When set to its default value of *false* (as recommended), the fit will optimize the variance to be minimal at the solution. If set to *true*, the covariance will be optimized instead.

Delphi: `property AverageX: Double read FAverageX;`
C++: `__property double AverageX = {read=FAverageX};`
- The arithmetic mean of the X-values.

Delphi: `property AverageY: Double read FAverageY;`
C++: `__property double AverageY = {read=FAverageY};`
- The arithmetic mean of the Y-values.

Delphi: `property VarianceX: Double read FVarianceX;`
C++: `__property double VarianceX = {read=FVarianceX};`
- The variance (σ) of the Y-values.

Delphi: `property Probability: Double read FProbability;`
C++: `__property double Probability = {read=FProbability};`
- The probability (r^2) of the function model fits to the empirical data points.

Delphi: `property SumOfResiduals: Double read FCurrentChiSquare;`
C++: `__property double SumOfResiduals = {read=FCurrentChiSquare};`
- The sum of the residuals (square deviations) from the calculated line to the measured data points.

Events

Delphi: `property FittingFunction: TRtFittingFunctionHandler read FFittingFkt write SetFittingFkt;`
C++: `__property TRtFittingFunctionHandler FittingFunction = {read=FFittingFkt, write=SetFittingFkt};`
- Your user function that returns the function value at a specified simplex corner (set of coefficients).

Delphi: `property OnNewCorner: TRtFittingCornerHandler read mNewCorner write mNewCorner;`
C++: `__property TRtFittingCornerHandler OnNewCorner = {read=mNewCorner, write=mNewCorner};`
- Event triggered each time a new simplex corner (set of coefficients) is generated for testing. Can be used to prepare pre-calculations for your ***FittingFunction***, which will be called directly afterwards for each X-value of the data vector supplied.

Delphi: `property OnIteration: TNotifyEvent read mDoDisplay write mDoDisplay;`
C++: `__property Classes::TNotifyEvent OnIteration = {read=mDoDisplay, write=mDoDisplay};`
- Triggered after each iteration. Can be used to display iteration results or to test if the fit should be continued or not.

Public Methods

Delphi: `function FromExpression: Boolean;`
C++: `bool FromExpression(void);`
- Returns *true* if no ***FittingFunction*** was defined.

Delphi: `function GetY(AtX: Double): Double;`
C++: `double GetY(double AtX);`
↳ Calculate the function result at a specified X-value using the coefficients with the minimum residuals.

Delphi: `procedure StopIterations;`
C++: `void StopIterations(void);`
↳ Breaks the optimization iterations loop; the iterations counter is not completed.

Delphi: `procedure Calculate; override;`
C++: `virtual void Calculate(void)`
↳ Calculate the simplex fit.

3.1.6 TRtInterpolationCalculation Class

This class is used to calculate interpolating third order polynomial coefficients between the data vector points. This class is used by the TRtInterpolation 34, TRtDifferential 34 and TRtIntegral 35 series line components. Nevertheless, it can be used separately to perform calculations without displaying any graph lines.

Unit/Namespace: `RtInterpolationCalculations`

Declarations

Delphi:
```
TRtInterpolationMethod=(Akima,CubicSpline,
ApproximativeSpline);
```
C++:
```
enum TRtInterpolationMethod { Akima, CubicSpline,
ApproximativeSpline };
```
↳ The different calculation methods supplied.

Delphi:
```
TRtDoublePoint = record
  X,Y: Double;
end;
```
C++:
```
struct TRtDoublePoint
{
public:
      double X;
      double Y;
};
```
↳ The record used to return minima and maxima.

Delphi: `TRtDoublePointArray = array of TRtDoublePoint;`
C++: `typedef DynamicArray<TRtDoublePoint > TRtDoublePointArray;`
↳ The array used to store the minima and maxima.

Delphi: `TRtInterpolationCalculation = class(TRtCalculation 44);`
C++:
```
class TRtInterpolationCalculation : public Rtfunctionseries::
TRtCalculation;
```

Public Properties

Delphi:
```
property InterpolationMethod: TRtInterpolationMethod read
FInterpolationMethod write SetInterpolationMethod;
```
C++:
```
__property TRtInterpolationMethod InterpolationMethod =
{read=FInterpolationMethod, write=SetInterpolationMethod,
nodefault};
```
↳ The method used for interpolation: Akima, Cubic Spline or Approximative Spline.

Delphi: `property SmoothPoints: Boolean read FDoSmoothing write SetDoSmoothing;`
C++: `__property bool SmoothPoints = {read=FDoSmoothing, write=SetDoSmoothing, nodefault};`
↳ If set to *true*, then the data points are smoothed using a linearly-weighted moving average (Akima and Cubic Spline only).

Delphi: `property SmoothingRange: Double read FSmoothM write SetSmoothM;`
C++: `__property double SmoothingRange = {read=FSmoothM, write=SetSmoothM};`
↳ The range used for the smoothing moving average (Akima and Cubic Spline only).

Delphi: `property ElasticityFactor: Double read FElasticityFactor write SetElasticityFactor;`
C++: `__property double ElasticityFactor = {read=FElasticityFactor, write=SetElasticityFactor};`
↳ The weighting factor that determines the effect of the data points to be passed (Approximative Spline only).

Delphi: `property Minima: TRtDoublePointArray read GetMinima;`
C++: `__property TRtDoublePointArray Minima = {read=GetMinima};`
↳ The local minima of the interpolation line.

Delphi: `property Maxima: TRtDoublePointArray read GetMaxima;`
C++: `__property TRtDoublePointArray Maxima = {read=GetMaxima};`
↳ The local maxima of the interpolation line.

Delphi: `property DifferentialMinima: TRtDoublePointArray read GetDiffMinima;`
C++: `__property TRtDoublePointArray DifferentialMinima = {read=GetDiffMinima};`
↳ The local minima of the differential of the interpolation line.

Delphi: `property DifferentialMaxima: TRtDoublePointArray read GetDiffMaxima;`
C++: `__property TRtDoublePointArray DifferentialMaxima = {read=GetDiffMaxima};`
↳ The local maxima of the differential of the interpolation line.

Delphi: `property IntegralMinima: TRtDoublePointArray read GetIntMinima;`
C++: `__property TRtDoublePointArray IntegralMinima = {read=GetIntMinima};`
↳ The local minima of the integral of the interpolation line.

Delphi: `property IntegralMaxima: TRtDoublePointArray read GetIntMaxima;`
C++: `__property TRtDoublePointArray IntegralMaxima = {read=GetIntMaxima};`
↳ The local maxima of the integral of the interpolation line.

Public Methods

Delphi: `function Interpolate(const X: Double): Double;`
C++: `double Interpolate(const double X);`
↳ Returns the interpolated value at a defined X-position.

Delphi: `function Slope(const X: Double): Double;`
C++: `double Slope(const double X);`
↳ Returns the interpolated slope (differential) at a defined X-position.

Delphi: `function Curvature(const X: Double): Double;`
C++: `double Curvature(const double X);`
↳ Returns the interpolated curvature (second differential) at a defined X-position.

Delphi: `function Integral(const X: Double): Double;`
C++: `double __fastcall Integral(const double X);`
↳ Returns the integral value at a defined X-position for the interpolated function.

Delphi: `procedure Calculate; override;`
C++: `virtual void Calculate(void)`
↳ Calculates the interpolation polynomial coefficients.

3.1.7 TRtMovingAverageCalculation Class

This class is used to calculate moving averages or trend lines using the data vectors. It is used by the TRtMovingAverage 33 series line component. Nevertheless, it can be used separately to perform calculations without showing any graph lines.

Unit/Namespace: `RtMovingAverage`

Declarations

Delphi: `TRtAveragingMethod=(Simple,LinearWeighted, ExponentiallyWeighted);`
C++: `enum TRtAveragingMethod { Simple, LinearWeighted, ExponentiallyWeighted };`
↳ The summing weighting method used in the calculation.

Delphi: `TRtAveragingDirection=(OnlyDownwards,UpAndDownwards);`
C++: `enum TRtAveragingDirection { OnlyDownwards, UpAndDownwards };`
↳ The direction used for the sum, either only downwards or upwards and downwards to the point of interest.

Delphi: `TRtAveragingRangeMethod=(NasCount,NasXDistance);`
C++: `enum TRtAveragingRangeMethod { NasCount, NasXDistance };`
↳ The method for finding the averaging range, either simply by count or by a defined X-range.

Delphi: `TRtMovingAverageCalculation = class(TRtCalculation 44);`
C++: `class TRtMovingAverageCalculation : public Rtfunctionseries::TRtCalculation;`

Public Variables

Delphi: `N,ExponentialFactorForN: Double;`
C++: `double N; double ExponentialFactorForN;`
↳ Constants used for calculation: the averaging range/count N and the factor used with exponential weighting.

Delphi: `AveragingMethod: TRtAveragingMethod;`
C++: `TRtAveragingMethod AveragingMethod;`
↳ The weighting method used for the sum: simple, linear or exponential.

Delphi: `AveragingDirection: TRtAveragingDirection;`
C++: `TRtAveragingDirection AveragingDirection;`
↳ Specifies whether the sum is taken only upwards, or upwards and downwards to the point of interest.

Delphi: `AveragingRangeMethod: TRtAveragingRangeMethod;`
C++: `TRtAveragingRangeMethod AveragingRangeMethod;`
↳ Specifies whether the range for the sum (***N***) is interpreted as count or as an X-distance.

Public Methods

Delphi: `function GetY(AtX: Double): Double;`
C++: `double GetY(double AtX);`
↳ Calculate the function result at a specified X-value.

Delphi: `procedure Calculate; override;`
C++: `virtual void Calculate(void)`
↳ Calculate the function parameters.

3.1.8 TRtDiscreteFourierCalculation

This class is used to calculate the Discrete Fourier Transformation of a source data set. This method is used internally by the TRtDiscreteFourier 130 series line component. Nevertheless, it can be used separately to perform calculations without displaying any graph lines.

Unit/Namespace: `RtFFT`

Declaration

Delphi: `TRtDiscreteFourierCalculation = class(TRtCalculation)` 44
C++: `class TRtDiscreteFourierCalculation : public Rtfunctionseries::TRtCalculation;`

Public Properties

Delphi: `property Amplitude: TRtDoubleVector read FAmplitude;`
C++: `__property Rtvectors::TRtDoubleVector* Amplitude = {read=FAmplitude};`
↳ The array of amplitudes of the result frequency factors.

Delphi: `property Phase: TRtDoubleVector read FPhase;`
C++: `__property Rtvectors::TRtDoubleVector* Phase = {read=FPhase};`
↳ The array of phases of the result frequency factors.

3.1.9 TRtFunctionParser Class

This class provides a small function parser. It is used by **TRtCaptions** 10, the property editor for **TRtDoubleVector** 22 and by the **TRtFittedLine** if the ***Expression*** 31 needs to be interpreted since no ***FittingFunction*** was supplied. Nevertheless, you may also find it useful for your own purposes.

Unit/Namespace: `RtParseFkt`

Declarations

Delphi:
```
TRtOnePrmFunction = function(One: Double): Double;
```
C++:
```
typedef double __fastcall (*TRtOnePrmFunction)(double One);
```
↳ Type used for functions using one parameter, such as sin().

Delphi:
```
TRtFunctionRec = record
  Id,Comment: string;
  AFunction: TRtOnePrmFunction;
end;
```
C++:
```
struct TRtFunctionRec{
public:
      AnsiString Id;
      AnsiString Comment;
      TRtOnePrmFunction AFunction;};
```
↳ Record stores the function in a list, which is parsed when the expression changes.

Delphi:
```
TRtFunctionsList = array of TRtFunctionRec;
```
C++:
```
typedef DynamicArray<TRtFunctionRec >  TRtFunctionsList;
```
↳ The list that stores the functions to be parsed.

Delphi:
```
TRtDoubleGetter = function: Double of object;
```
C++:
```
typedef double __fastcall ( __closure *TRtDoubleGetter)(void);
```
↳ A function that returns a double. Used to retrieve constants such as pi.

Delphi:
```
TRtDoubleGetterRec = record
  Id,Comment: string;
  Container: TObject;
  AGetter: TRtDoubleGetter;
end;
```
C++:
```
struct TRtDoubleGetterRec
{
public:
      AnsiString Id;
      AnsiString Comment;
      System::TObject* Container;
      TRtDoubleGetter AGetter;
};
```
↳ The record storing the constant. To be used by the parser.

Delphi:
```
TRtDoubleGetterList = array of TRtDoubleGetterRec;
```
C++:
```
typedef DynamicArray<TRtDoubleGetterRec>
TRtDoubleGetterList;
```
↳ The list storing the constants for the parser.

Delphi:
```
TRtFunctionParser = class;
```
C++:
```
class TRtFunctionParser : public System::TObject
```

Public Variables

Delphi: `FunctionsList: TRtFunctionsList;`
C++: `DynamicArray<TRtFunctionRec> FunctionsList;`
↳ Can be used to extend the parser with your own functions.

Delphi: `VariablesList: TRtDoubleGetterList;`
C++: `DynamicArray<TRtDoubleGetterRec> VariablesList;`
↳ Can be used to extend the parser with your own constants.

Public Properties

Delphi: `property Expression: string read FExpression write SetExpression;`
C++: `__property AnsiString Expression = {read=FExpression, write=SetExpression};`
↳ The expression is parsed if you write to the property. The internal functions binary tree persists and can therefore return the result as many times as are needed.

Delphi: `property Result: Double read GetResult;`
C++: `__property double Result = {read=GetResult};`
↳ Returns the result of the expression.

Methods

Delphi: `procedure AddFunctionToList(Id,Comment: string; AFunction: TRtOnePrmFunction; var AList: TRtFunctionsList);`
C++: `extern PACKAGE void __fastcall AddFunctionToList(AnsiString Id, AnsiString Comment, TRtOnePrmFunction AFunction, TRtFunctionsList &AList);`
↳ Adds a one parameter function to the parser list.

Delphi: `procedure RemoveFromFunctionsList(Id: string; var AList: TRtFunctionsList);`
C++: `extern PACKAGE void __fastcall RemoveFromFunctionsList (AnsiString Id, TRtFunctionsList &AList);`
↳ Removes the entry from the parser list.

Delphi: `procedure AddVariableToList(Id,Comment: string; Container: TObject; AGetter: TRtDoubleGetter; var AList: TRtDoubleGetterList);`
C++: `extern PACKAGE void __fastcall AddVariableToList(AnsiString Id, AnsiString Comment, System::TObject* Container, TRtDoubleGetter AGetter, TRtDoubleGetterList &AList);`
↳ Adds a constant return function to the parser list.

Delphi: `procedure RemoveFromVariablesList(Id: string; var AList: TRtDoubleGetterList);`
C++: `extern PACKAGE void __fastcall RemoveFromVariablesList (AnsiString Id, TRtDoubleGetterList &AList);`
↳ Removes the entry from the parser list.

3.1.10 TRtPropertyLinks Class

The **TRtPropertyLinks** class was developed to provide an easy way of setting properties with controls in the "Rt-Tools2D Standard" palette, without needing to use ***OnChanged*** or ***OnSelect*** events.

Unit/Namespace: `RtPropertyLink`

Declaration

Delphi: `TRtPropertyLinks = class(TPersistent)`

C++: `class TRtPropertyLinks : public Classes::TPersistent`

Public Properties

Delphi: `property TypeFilter:string read FTypeFilter write FTypeFilter;`

C++: `__property AnsiString TypeFilter = {read=FTypeFilter, write=FTypeFilter};`

Name of the property type. On class creation, this property type is automatically set using RTTI information to the type of the ManagedProperty 56. This is used in the property editor for example to filter the items shown in the tree view so that only relevant items are displayed. For filtering purposes, TypeKind 56 is primarily used. As an example, the **TRtDoubleEdit** component with a managed property of type **double** sets this property to a blank string to additionally enable the linking of **Single** properties.

Delphi: `property TypeKind:TTypeKind read FTypeKind;`

C++: `__property Typinfo::TTypeKind TypeKind = {read=FTypeKind, nodefault};`

On class creation this property is automatically set using RTTI information to the type of the ManagedProperty 56. Used internally within the property tree of the property editor.

Delphi: `property ManagedProperty:string read FManagedProperty;`

C++: `__property AnsiString ManagedProperty = {read=FManagedProperty};`

The name of the ManagedProperty 57 from the creation of the class (for internal use only).

Delphi: `property Items: TStrings read FItems write SetItems;`

C++: `__property Classes::TStrings* Items = {read=FItems, write=SetItems};`

The list of linked properties described, with the form name in root position in the full name. Example: "Form1.Label1.Font.Color". This is normally set with the special property editor, but can also be changed programmatically. Ensure you call Modified 57 and CheckLink 57 afterwards to ensure the smooth functioning of your program.

Public Methods

Delphi: `constructor Create( ManagedInstance: TComponent; ManagedProperty: string); virtual;`
C++: `virtual TRtPropertyLinks( Classes::TComponent* ManagedInstance, AnsiString ManagedProperty);`

This constructor is used in the "Rt-Tools2D Standard" controls to create a internal link class. The ***ManagedInstance*** is the calling component, and the ***ManagedProperty*** is the name of the property in the calling component, providing the value to be set for all the components contained in the linked Items list.

Delphi: `procedure UpdateLinkNames;`
C++: `void UpdateLinkNames( void);`

Updates the internal list of linked items (for internal use only).

Delphi: `procedure CheckLink;`
C++: `void CheckLink( void);`

Is called internally after loading the list of items, building up internal structures using the RTTI information of the listed components and properties. Also sets the preset value of the ManagedProperty 56 to the value of the property in Items 56[0].

Delphi: `procedure Modified;`
C++: `void Modified( void);`

If you are changing the items list programmatically then you must call Modified CheckLink 57 afterwards to ensure the smooth running of your program.

Delphi:
```
procedure SetLink( Value: Integer); overload;
procedure SetLink( Value: TRtColor 58); overload;
procedure SetLink( Value: Boolean); overload;
procedure SetLink( Value: Double); overload;
procedure SetLink( Value: TRtRichCaption 70); overload;
procedure SetLink( Value: TRtDashStyle 58); overload;
procedure SetLink( Value: TRtAreaStyle 59); overload;
procedure SetLink( Value: TRtPointSymbol 63); overload;
```
C++:
```
void SetLink( int Value);
void SetLink( Rtgdi::TRtColor Value);
void SetLink( bool Value);
void SetLink( double Value);
void SetLink( WideString Value);
void SetLink( Gdipapi::TDashStyle Value);
void SetLink( Rtgdi::TRtAreaStyle Value);
void SetLink( Rtdrawing::TRtPointSymbol Value);
```

Used in the "Rt-Tools2D Standard" controls to set the linked properties when the ManagedProperty 56 changes.

3.1.11 The RtGDI Unit

This unit is needed to interface with the fundamental Windows GDI+ drawing routines. RtTools2D does not use the standard Delphi TCanvas drawing routines but uses either the GDPlus library from http://www.progdigy.com for Win32 applications or the System.Drawing and System.Drawing.Drawing2D namespaces for VCL.net applications.

This unit creates classes to interface with both programming platforms using the same names.

Unit/Namespace: `RtGDI`

Declarations

Delphi:
```
TRtColor = type Integer;            // VCL.net
TRtColor = type Gdipapi.TGPColor;   // Win32
```
C++:
```
typedef unsigned TRtColor;
```
↳ Type used for all colors. Although the source types are identical to the standard **TColor**, they are used differently. **TRtColor** uses GDI+ Alpha-RGB byte order to display the colors. Conversion functions in both directions are supplied.

Delphi:
```
TRtDashStyle = System.Drawing.Drawing2D.DashStyle;  // VCL.net
TRtDashStyle = Gdipapi.TDashStyle;                  // Win32
```
C++:
```
typedef Gdipapi::TDashStyle TRtDashStyle;
```
↳ Type used for line dash styles.

Delphi:
```
TRtPen = System.Drawing.Pen;            // VCL.net
TRtPen = class(Gdipobj.TGPPen);         // Win32
```
C++:
```
class TRtPen : public Gdipobj::TGPPen;
```
↳ Type used for line drawing.

Delphi:
```
TRtBrush = System.Drawing.Brush;  // VCL.net
TRtBrush = Gdipobj.TGPBrush;      // Win32
```
C++:
```
typedef TGPBrush TRtBrush;
```
↳ Type used for areas drawn below lines.

Delphi:
```
TRtSolidBrush = System.Drawing.SolidBrush;  // VCL.net
TRtSolidBrush = Gdipobj.TGPSolidBrush;      // Win32
```
C++:
```
typedef TGPSolidBrush TRtSolidBrush;
```
↳ Type used for solid filled areas drawn below lines.

Delphi:
```
TRtPathGradientBrush = System.Drawing.Drawing2D.
PathGradientBrush; // VCL.net
TRtPathGradientBrush = Gdipobj.TGPPathGradientBrush;
      // Win32
```
C++:
```
typedef TGPPathGradientBrush TRtPathGradientBrush;
```
↳ Type used for drawing areas.

Delphi:
```
TRtHatchBrush = System.Drawing.HatchBrush;  // VCL.net
TRtHatchBrush = Gdipobj.TGPHatchBrush;      // Win32
```
C++:
```
typedef TGPHatchBrush TRtHatchBrush;
```
↳ Type used for hatch filled areas drawn below lines.

Delphi:
```
TRtLinearGradientBrush = System.Drawing.Drawing2D.
LinearGradientBrush; // VCL.net
TRtLinearGradientBrush = Gdipobj.TGPLinearGradientBrush;
          // Win32
```
C++:
```
typedef TGPLinearGradientBrush TRtLinearGradientBrush;
```
↳ Type used for drawing areas.

Delphi:
```
TRtGraphics = System.Drawing.Graphics;  // VCL.net
TRtGraphics = Gdipobj.TGPGraphics;      // Win32
```
C++:
```
typedef TGPGraphics TRtGraphics;
```
↳ The drawing object class used as the target for all drawing API operations. Used similarly as a TCanvas.

Delphi: `TRtImage = System.Drawing.Image; // VCL.net`
`TRtImage = Gdipobj.TGPImage; // Win32`
C++: `typedef TGPImage TRtImage;`
↳ Type used for drawing bitmap and metafile images.

Delphi: `TRtPoint = System.Drawing.PointF; // VCL.net`
`TRtPoint = Gdipapi.TGPPointF; // Win32`
C++: `typedef Gdipapi::TGPPointF TRtPoint;`
↳ Type used to store single X/Y point records.

Delphi: `TRtPointArray = array of System.Drawing.PointF; // VCL.net`
`TRtPointArray = Gdipapi.TPointFDynArray; // Win32`
C++: `typedef DynamicArray<Gdipapi::TGPPointF> TRtPointFArray;`
↳ Array of points. Used to draw poly lines.

Delphi: `TRtRectangle = System.Drawing.Rectangle; // VCL.net`
`TRtRectangle = Gdipapi.TGPRect; // Win32`
C++: `typedef Gdipapi::TGPRect TRtRectangle;`
↳ Type used to draw rectangles with **integer** corners.

Delphi: `TRtRectangleF = System.Drawing.RectangleF; // VCL.net`
`TRtRectangleF = Gdipapi.TGPRectF; // Win32`
C++: `typedef Gdipapi::TGPRectF TRtRectangleF;`
↳ Type used to draw rectangles with **Single** corners.

Delphi:
```
TRtAreaStyle = (Horizontal, Vertical, ForwardDiagonal,
BackwardDiagonal, Cross, DiagonalCross, Percent05, Percent10,
Percent20, Percent25, Percent30, Percent40, Percent50,
Percent60, Percent70, Percent75, Percent80, Percent90,
LightDownwardDiagonal, LightUpwardDiagonal,
DarkDownwardDiagonal, DarkUpwardDiagonal,
WideDownwardDiagonal, WideUpwardDiagonal, LightVertical,
LightHorizontal, NarrowVertical, NarrowHorizontal,
DarkVertical, DarkHorizontal, DashedDownwardDiagonal,
DashedUpwardDiagonal, DashedHorizontal, DashedVertical,
SmallConfetti, LargeConfetti, ZigZag, Wave, DiagonalBrick,
HorizontalBrick, Weave, Plaid, Divot, DottedGrid,
DottedDiamond, Shingle, Trellis, Sphere, SmallGrid,
SmallCheckerBoard, LargeCheckerBoard, OutlinedDiamond,
SolidDiamond, Total);
```
C++:
```
typedef TRtAreaStyle { Horizontal, Vertical, ForwardDiagonal,
BackwardDiagonal, Cross, DiagonalCross, Percent05, Percent10,
Percent20, Percent25, Percent30, Percent40, Percent50,
Percent60, Percent70, Percent75, Percent80, Percent90,
LightDownwardDiagonal, LightUpwardDiagonal,
DarkDownwardDiagonal, DarkUpwardDiagonal,
WideDownwardDiagonal, WideUpwardDiagonal, LightVertical,
LightHorizontal, NarrowVertical, NarrowHorizontal,
DarkVertical, DarkHorizontal, DashedDownwardDiagonal,
DashedUpwardDiagonal, DashedHorizontal, DashedVertical,
SmallConfetti, LargeConfetti, ZigZag, Wave, DiagonalBrick,
HorizontalBrick, Weave, Plaid, Divot, DottedGrid,
DottedDiamond, Shingle, Trellis, Sphere, SmallGrid,
SmallCheckerBoard, LargeCheckerBoard, OutlinedDiamond,
SolidDiamond, Total };
```
↳ Type used for drawing hatch-filled areas.

Methods

Delphi: `function GetAlpha( Color: TRtColor 58 : Byte;`
C++: `extern PACKAGE Byte __fastcall GetAlpha( TRtColor Color);`
Returns the alpha (transparency) byte of the ARGB color.

Delphi: `function GetR( Color: TRtColor 58 : Byte;`
C++: `extern PACKAGE Byte __fastcall GetR( TRtColor Color);`
Returns the red byte of the ARGB color.

Delphi: `function GetG( Color: TRtColor 58 : Byte;`
C++: `extern PACKAGE Byte __fastcall GetG( TRtColor Color);`
Returns the green byte of the ARGB color.

Delphi: `function GetB( Color: TRtColor 58 : Byte;`
C++: `extern PACKAGE Byte __fastcall GetB( TRtColor Color);`
Returns the blue byte of the ARGB color.

Delphi: `function GetBrightness( Color: TRtColor 58 : Single;`
C++: `extern PACKAGE float __fastcall GetBrightness( TRtColor Color);`
Returns the brightness value of the ARGB color (0 = black; 100 = white).

Delphi: `function GetHue( Color: TRtColor 58 : Single;`
C++: `extern PACKAGE float __fastcall GetHue( TRtColor Color);`
Returns the color hue (position in degrees on the color circle) of the ARGB color (0 = red; 60 = yellow; 120 = green; 180 = cyan; 240 = blue; 320 = magenta).

Delphi: `function GetSaturation( Color: TRtColor 58 : Single;`
C++: `extern PACKAGE float __fastcall GetSaturation( TRtColor Color);`
Returns the saturation value of the ARGB color (50 = full color; >50 = color grayed to white; <50 = color grayed to black).

Delphi: `function TColorToARGB( Color: TColor): TRtColor 58;`
C++: `extern PACKAGE TRtColor __fastcall TColorToARGB( Graphics:: TColor Color)`
Converts a standard Delphi TColor to the corresponding ARGB color.

Delphi: `function ARGBToTColor( Color: TRtColor 58 : TColor;`
C++: `extern PACKAGE Graphics::TColor __fastcall ARGBToTColor ( TRtColor Color);`
Converts an ARGB color to the corresponding standard Delphi TColor.

Delphi:
```
function ColorToARGB( A, R, G, B: Byte): TRtColor 58; overload;
function ColorToARGB( Color: string): TRtColor; overload;
```
C++:
```
extern PACKAGE TRtColor __fastcall ColorToARGB( Byte A, Byte R, Byte G, Byte B);
extern PACKAGE TRtColor __fastcall ColorToARGB( AnsiString Color);
```
Returns an ARGB color, supplying either the A,R,G,B component bytes or one of the known color names, such as Transparent, White, Blue, etc.

Known Colors

System:

ActiveBorder	ActiveCaption	ActiveCaptionText	AppWorkspace	Control
ControlDark	ControlDarkDark	ControlLight	ControlLightLight	ControlText
Desktop	GrayText	Highlight	HighlightText	HotTrack
InactiveBorder	InactiveCaption	InactiveCaptionText	Info	InfoText
Menu	MenuText	ScrollBar	Window	WindowFrame
WindowText				

Other:

Transparent	Black	DimGray	Gray	DarkGray
Silver	LightGray	Gainsboro	WhiteSmoke	White
RosyBrown	IndianRed	Brown	Firebrick	LightCoral
Maroon	DarkRed	Red	Snow	MistyRose
Salmon	Tomato	DarkSalmon	Coral	OrangeRed
LightSalmon	Sienna	SeaShell	Chocolate	SaddleBrown
SandyBrown	PeachPuff	Peru	Linen	Bisque
DarkOrange	BurlyWood	Tan	AntiqueWhite	NavajoWhite
BlanchedAlmond	PapayaWhip	Moccasin	Orange	Wheat
OldLace	FloralWhite	DarkGoldenrod	Goldenrod	Cornsilk
Gold	Khaki	LemonChiffon	PaleGoldenrod	DarkKhaki
Beige	LightGoldenrodYellow	Olive	Yellow	LightYellow
Ivory	OliveDrab	YellowGreen	DarkOliveGreen	GreenYellow,
Chartreuse	LawnGreen	DarkSeaGreen	ForestGreen	LimeGreen
LightGreen	PaleGreen	DarkGreen	Green	Lime
Honeydew	SeaGreen	MediumSeaGreen	SpringGreen	MintCream
MediumSpringGreen	MediumAquamarine	Aquamarine	Turquoise	LightSeaGreen
MediumTurquoise	DarkSlateGray	PaleTurquoise	Teal	DarkCyan
Aqua	Cyan	LightCyan	Azure	DarkTurquoise
CadetBlue	PowderBlue	LightBlue	DeepSkyBlue	SkyBlue
LightSkyBlue	SteelBlue	AliceBlue	DodgerBlue	SlateGray
LightSlateGray	LightSteelBlue	CornflowerBlue	RoyalBlue	MidnightBlue
Lavender	Navy	DarkBlue	MediumBlue	Blue
GhostWhite	SlateBlue	DarkSlateBlue	MediumSlateBlue	MediumPurple
BlueViolet	Indigo	DarkOrchid	DarkViolet	MediumOrchid
Thistle	Plum	Violet	Purple	DarkMagenta
Magenta	Fuchsia	Orchid	MediumVioletRed	DeepPink
HotPink	LavenderBlush	PaleVioletRed	Crimson	Pink
LightPink				

Delphi: `function ARGBToString(Color: TRtColor; WithSysColors: Boolean = True): string;`

C++: `extern PACKAGE AnsiString __fastcall ARGBToString(TRtColor Color, bool WithSysColors = true);`

Returns the known color name of the ARGB color. If WithSysColors is set to true, known system colors such as Control, Highlight or Window will also be provided in the output. If no known color name matches the color, the output will be the ARGB integer given as a hex number starting with '$'.

Delphi: `function ARGBToRichID(Color: TRtColor): string;`

C++: `extern PACKAGE AnsiString __fastcall ARGBToRichID(TRtColor Color);`

Returns the components of the ARGB color as string. Red for example = '[A=255, R=255, G=0, B=0]'. Used for color changes in captions.

Delphi: `function RGBToRichID(Color: TColor): string;`
C++: `extern PACKAGE AnsiString __fastcall RGBToRichID(Graphics::TColor Color)`

↳ Returns the components of the TColor as a string. Red for example = '[A=255, R=255, G=0, B=0]'.

Delphi: `function RichIDToTRtColor(Color: string): TRtColor;` 58
C++: `extern PACKAGE TRtColor __fastcall RichIDToTRtColor(AnsiString Color);`

↳ Convert a color components string such as '[A=255, R=255, G=0, B=0]' (= red) to the corresponding ARGB color.

Delphi: `function RichIDToTColor(Color: string): TColor;`
C++: `extern PACKAGE TColor __fastcall RichIDToTColor(AnsiString Color);`

↳ Convert a color components string such as '[A=255, R=255, G=0, B=0]' (= red) to the corresponding TColor.

Delphi: `function ContrastBackGround(Color: TColor): TRtColor;` 58
C++: `extern PACKAGE TRtColor __fastcall ContrastBackGround(Graphics::TColor Color);`

↳ Returns an ARGB color that contrasts well with the supplied TColor. Used to get background colors for edit controls.

Delphi: `function ContrastBackGroundARGB(Color: TRtColor): TRtColor;` 58
C++: `extern PACKAGE TRtColor __fastcall ContrastBackGroundARGB(TRtColor Color);`

↳ Returns an ARGB color that contrasts well with the supplied ARGB color. Used to get background colors for style selection controls.

Delphi: `function ContrastForeColorARGB(Color: TRtColor): TRtColor;` 58
C++: `extern PACKAGE TRtColor __fastcall ContrastForeColorARGB(TRtColor Color);`

↳ Returns an ARGB color that contrasts well with the supplied ARGB color. Used to get foreground colors for range selection lines.

Delphi: `function HSLToColor(const H,S,L: Single): TRtColor;` 58
C++: `extern PACKAGE HSLToColor(const float H, const float S, const float L);`

↳ Convert hue, saturation and luminance(brightness) to an ARGB color.

Delphi: `function Brighten(Value: TRtColor; Factor: Single): TRtColor;` 58
C++: `extern PACKAGE Byte __fastcall Brighten(TRtColor Value, float Factor);`

↳ Brighten the color by the factor given, meaning multiply the brightness by the factor. If the factor is <1 this will darken the color.

Delphi: `function GetAreaBrush(Style: TRtAreaStyle; ForeColor, BackColor: TRtColor): TRtBrush;` 59 58 58
C++: `extern PACKAGE Gdipobj_rt::TGPBrush* __fastcall GetAreaBrush(TRtAreaStyle Style, TRtColor ForeColor, TRtColor BackColor);`

↳ Return the brush using the colors and area style. If ***Style*** is set to *Total* the result will be of type TRtSolidBrush 58, otherwise of type TRtHatchBrush 58.

3.1.12 The RtDrawing Unit

This unit implements some drawing routines using the GDI+ graphics 58. Methods are provided for drawing point symbols, bars, cylinders and rounded rectangles.

Unit/Namespace: `RtDrawing`

Declaration

Delphi:
```
TRtPointSymbol=(Dot,Cross,Ex,Star,FiveStar,Circle,Square,
Diamond,Delta,Nabla);
```
C++:
```
enum TRtPointSymbol { Dot, Cross, Ex, Star, FiveStar, Circle,
Square, Diamond, Delta, Nabla };
```
↳ Type used to draw different point symbols.

Methods

Delphi:
```
procedure DrawPointSymbol(Gr: TRtGraphics 58; Border: TRtPen 58
; Fill: TRtBrush 58; const X,Y,Size: Single; Symbol:
TRtPointSymbol 63);
```
C++:
```
extern PACKAGE void __fastcall DrawPointSymbol(Gdipobj::
TGPGraphics* Gr, Rtgdi::TRtPen* Border, Gdipobj::TGPBrush*
Fill, const float X, const float Y, const float Size,
TRtPointSymbol Symbol);
```
↳ Draws a point symbol at defined X,Y position and with a defined size, using the pen for the border line and the fill brush.

Delphi:
```
procedure ExcludeSymbolSpace(Gr: TRtGraphics 58; const X,Y,
Size: Single; Symbol: TRtPointSymbo 63L);
```
C++:
```
extern PACKAGE void __fastcall ExcludeSymbolSpace(Gdipobj::
TGPGraphics* Gr, const float X, const float Y, const float
Size, TRtPointSymbol Symbol);
```
↳ Used to add the space used by the point symbol to the graph clipping range. Needed when drawing series lines while leaving space for point symbols with transparent fills.

Delphi:
```
function OrderedRect(Left,Top,Right,Bottom: Integer): TRect;
overload;
function OrderedRect(Left,Top,Right,Bottom: Single):
TRtRectangleF 59; overload;
```
C++:
```
extern PACKAGE Types::TRect __fastcall OrderedRect(int Left,
int Top, int Right, int Bottom);
extern PACKAGE Gdipapi::TGPRectF __fastcall OrderedRect(float
Left, float Top, float Right, float Bottom);
```
↳ Returns a rectangle with the correct dimensions even though the order of the corners is twisted – this means that this also works where Left>Right or Bottom<Top.

Delphi:
```
procedure Draw3DBar(Gr: TRtGraphics 58; Line: TRtPen 58;
FillColor: TRtColor; const X1,Y1,X2,Y2,dX,dY: Single);
```
C++:
```
extern PACKAGE void __fastcall Draw3DBar(Gdipobj::
TGPGraphics* Gr, Rtgdi::TRtPen* Line, Rtgdi::TRtColor
FillColor, const float X1, const float Y1, const float X2,
const float Y2, const float dX, const float dY);
```
↳ Draws a 3D bar using the line for the edges and the ***FillColor*** to fill the face. The right side is darkened 62 while the top side is brightened 62. The ***X1***, ***Y1***, ***X2*** and ***Y2*** values give the corners of the front, while the ***dX*** and ***dY*** give the shifts in depth.

Delphi: `procedure DrawCylinderVertical(Gr: TRtGraphics; Line: TRtPen; FillColor: TRtColor; const X1,Y1,X2,Y2,dX,dY: Single);`

C++: `extern PACKAGE void __fastcall DrawCylinderVertical(Gdipobj::TGPGraphics* Gr, Rtgdi::TRtPen* Line, Rtgdi::TRtColor FillColor, const float X1, const float Y1, const float X2, const float Y2, const float dX, const float dY);`

Draws a vertically-aligned cylinder using the ***Line*** pen for the edges and the ***FillColor*** to fill the face. The 3D effect is generated by shading the fill color. The ***X1, Y1, X2*** and ***Y2*** values give the dimension, the ***dX***, ***dY*** the depth of the ellipse drawn at the top.

Delphi: `procedure DrawCylinderHorizontal(Gr: TRtGraphics; Line: TRtPen; FillColor: TRTColor; const X1,Y1,X2,Y2,dX,dY: Single);`

C++: `extern PACKAGE void __fastcall DrawCylinderHorizontal(Gdipobj::TGPGraphics* Gr, Rtgdi::TRtPen* Line, Rtgdi::TRtColor FillColor, const float X1, const float Y1, const float X2, const float Y2, const float dX, const float dY);`

Draws a horizontally-aligned cylinder using the ***Line*** pen for the edges and the ***FillColor*** to fill the face. The 3D effect is generated by shading the fill color. The ***X1, Y1, X2*** and ***Y2*** values give the dimension, the ***dX***, ***dY*** the depth of the ellipse drawn at the right.

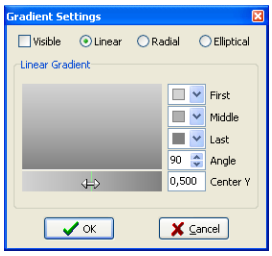
Delphi: `procedure DrawRoundRect(Gr: TRtGraphics; Fill: TRtBrush; Border: TRtPen; Rect: TRtRectangleF; Rounding: Single; Frame: Boolean = True; DropShadow: Boolean = False; Raised: Boolean = False; Lowered: Boolean = False);`

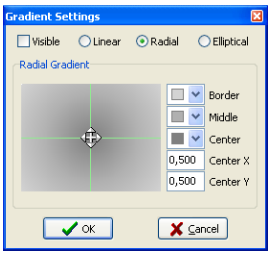
C++: `extern PACKAGE void __fastcall DrawRoundRect(Gdipobj_rt::TGPGraphics* Gr, Gdipobj_rt::TGPBrush* Fill, Rtgdi::TRtPen* Border, const Gdipapi_rt::TGPRectF &Rect, float Rounding, bool Frame = true, bool DropShadow = false, bool Raised = false, bool Lowered = false);`

Draws a rounded rectangle using the ***Border*** pen for the outline and the ***Fill*** brush. The ***Rounding*** defines the radius of the rounding at the edges. If ***Frame*** is set *true* the rectangle will have a frame border. If ***DropShadow*** is set to *true* a shadow will be drawn below the rounded rectangle. If ***Raised*** is set to *true* the rectangle is drawn raised. If ***Lowered*** is set to *true* the rectangle is drawn lowered.

3.2 Label and Edits

3.2.1 TRtGradientSettings Class





This class is used to store the properties of gradient backgrounds of captions the graph and its data area, the legend, line fills and areas drawn below lines, including their colors etc. enabling a more structured access. The properties can be set interactively using a special property editor.

Unit/Namespace: `RtStyles`

Declaration

Delphi: `TRtGradientSettings = class(TPersistent);`
C++: `class TRtGradientSettings : public Classes::TPersistent;`

Published Properties

Delphi: `property Visible: Boolean read FVisible write SetVisible;`
C++: `__property bool Visible = {read=FVisible, write=SetVisible, nodefault};`
The visibility of the gradient.

Delphi: `TRtGradientStyle = (Linear, Radial, Elliptical);`
`property Style: TRtGradientStyle read FGradientStyle write SetGradientStyle;`
C++: `enum TRtGradientStyle { Linear, Radial, Elliptical };`
`__property Rtgdi::TRtGradientStyle Style = {read=FGradientStyle, write=SetGradientStyle, nodefault};`
The drawing style of the gradient (linear, radial or elliptical).

Delphi: `property FirstColor: TRtColor` 58 `read FFirstColor write SetFirstColor;`
C++: `__property Rtgdi::TRtColor FirstColor = {read=FFirstColor, write=SetFirstColor, nodefault};`
The first color of the linear, radial or elliptical gradient. The gradient uses a 3 colors blend using FirstColor, MiddleColor 65 and LastColor 65.

Delphi: `property MiddleColor: TRtColor` 58 `read FMiddleColor write SetMiddleColor;`
C++: `__property Rtgdi::TRtColor MiddleColor = {read=FMiddleColor, write=SetMiddleColor, nodefault};`
The middle color of the linear, radial or elliptical gradient. The gradient uses a 3 colors blend using FirstColor 65, MiddleColor and LastColor 65.

Delphi: `property LastColor: TRtColor` 58 `read FLastColor write SetLastColor;`
C++: `__property Rtgdi::TRtColor LastColor = {read=FLastColor, write=SetLastColor, nodefault};`
The last color of the linear, radial or elliptical gradient. The gradient uses a 3 colors blend using FirstColor 65, MiddleColor 65 and LastColor.

Delphi: `property LinearAngle: Single read FLinearAngle write SetLinearAngle;`
C++: `__property float LinearAngle = {read=FLinearAngle, write=SetLinearAngle, nodefault};`
The angle in degrees to draw the linear gradient (0..360).

Delphi: `property LinearCenter: Single read FLinearCenter write SetLinearCenter;`
C++: `__property float LinearCenter = {read=FLinearCenter, write=SetLinearCenter, nodefault};`
The center for the middle color for the gradient (0..1). A value of 0.5 means center of the gradient rectangle.

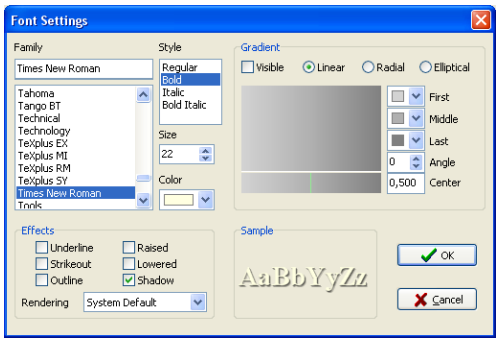
Delphi: `property RadialCenterX: Single read GetRadialCenterX write SetRadialCenterX;`
C++: `__property float RadialCenterX = {read=GetRadialCenterX, write=SetRadialCenterX, nodefault};`

↳ The X-component of the center of the last color for the gradient (0..1). A value of 0.5 means center of the gradient rectangle.

Delphi: `property RadialCenterY: Single read GetRadialCenterY write SetRadialCenterY;`
C++: `__property float RadialCenterY = {read=GetRadialCenterY, write=SetRadialCenterY, nodefault};`

↳ The Y-component of the center of the last color for the gradient (0..1). A value of 0.5 means center of the gradient rectangle.

3.2.2 TRtFont Class



This class enhances the functionality of TFont to enable extended drawing effects possible using GDI+ drawing. It is used for all text drawing in the Cartesian graph components. The properties can be set interactively using a special property editor.

Unit/Namespace: `RtRichLabel`

Declaration

Delphi: `TRtFont = class(TFont)`
C++: `__property TRtRichCaption PlainText = {read=GetPlainText};`

Published Properties

Delphi: `property Name: TFontName read GetName write SetName;`
C++: `__property Graphics::TFontName Name = {read=GetName, write=SetName};`

↳ The name to specify the typeface of the font.

Delphi: `property Style: TFontStyles read GetStyle write SetStyle`
C++: `__property Graphics::TFontStyles Style = {read=GetStyle, write=SetStyle, nodefault};`

↳ Determines whether the font is normal, italic, underlined, bold or any combination of it.

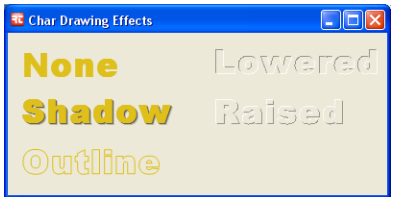
Delphi: `property Size: Integer read GetSize write SetSize stored False;`
C++: `__property int Size = {read=GetSize, write=SetSize, stored=false, nodefault};`

↳ The point size of the font.

Delphi: `property FillColor: TRtColor` 58 `read FFillColor write SetFillColor;`
C++: `__property Rtgdi::TRtColor FillColor = {read=FFillColor, write=SetFillColor, nodefault};`

↳ The fill color of the font characters. Note: The color supports transparency.

Delphi:
```
TRtCharDrawEffect = (Shadow, Outline, Lowered, Raised);
TRtCharDrawEffects = set of TRtCharDrawEffect;
property DrawEffects: TRtCharDrawEffects read FDrawEffects
write SetDrawEffects default [];
```
C++:
```
enum TRtCharDrawEffect { Shadow, Outline, Lowered, Raised };
typedef Set<TRtCharDrawEffect, Shadow, Raised>
TRtCharDrawEffects;
__property TRtCharDrawEffects DrawEffects =
{read=FDrawEffects, write=SetDrawEffects, default=0};
```
This property allows to draw the captions using special drawing effects as drop shadow, outline, lowered and raised.



Delphi:
```
property Gradient: TRtGradientSettings read FGradient write
GetGradient;
```
C++:
```
__property Rtstyles::TRtGradientSettings* Gradient =
{read=FGradient, write=SetGradient};
```
Settings for an optional gradient to fill the characters of the captions.

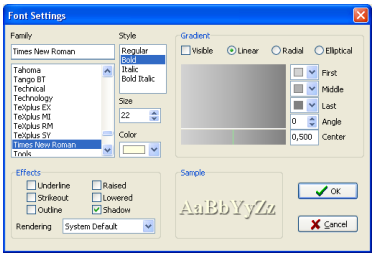
Delphi:
```
TRtTextRenderingHint = (SystemDefault,
SingleBitPerPixelGridFit, SingleBitPerPixel,
AntiAliasGridFit, AntiAlias, ClearTypeGridFit);
property RenderingHint: TRtTextRenderingHint read
FRenderingHint write SetRenderingHint default SystemDefault;
```
C++:
```
__property Graphics::TFontName Name = {read=GetName,
write=SetName};
```
The characters can be rendered to the screen different ways fitting to the screen pixel grid anti aliased or clear as clear type. We recommend to keep the system setting default.

Public Property

Delphi:
```
property AsTFont: TFont read GetAsTFont write SetAsTFont;
```
C++:
```
__property Graphics::TFont* AsTFont = {read=GetAsTFont,
write=SetAsTFont};
```
The TRtFont object can be converted to a TFont object e.g. to be used by the standard TFontDialog.

3.2.3 TRtFontDialog Class

This class provides a replacement for the standard Windows font selection dialog. It supports setting the enhanced drawing capabilities of TRtFont 66.



Unit/Namespace: `RtFontDialog`

Declaration

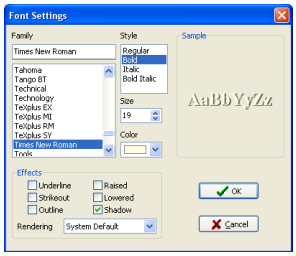
Delphi: `TRtFontDialog = class(TComponent);`
C++: `class TRtFontDialog : public Classes::TComponent;`

Published Properties

Delphi: `property Font: TRtFont` 66 `read FFont write SetFont;`
C++: `__property Rtrichlabel::TRtFont* Font = {read=FFont, write=SetFont};`

- The font to be set with the dialog.

Delphi: `property ShowGradientSettings: Boolean read FShowGradient write FShowGradient;`
C++: `__property bool ShowGradientSettings = {read=FShowGradient, write=FShowGradient, nodefault};`

- 

The default setting *true* will display the dialog including gradient settings. If you want the dialog to show only normal color fill you can set to *false*.

Delphi: `property ShowApply: Boolean read FShowApply write FShowApply;`
C++: `__property bool ShowApply = {read=FShowApply, write=FShowApply, nodefault};`

- If you set this property to *true* the dialog will contain an Apply button. If the user clicks tis button the OnApply 69 event will be triggered. This can be used to apply the current settings without closing the dialog.

Delphi: `property ShowHelp: Boolean read FShowHelp write FShowHelp;`
C++: `__property bool ShowHelp = {read=FShowHelp, write=FShowHelp, nodefault};`

- If you set this property to *true* the dialog will contain a Help button. If the user clicks tis button the help system will open the relevant help page. The help system will also be opened when pressing the F1 key when the dialog is displayed. Any case if the Help button is visible or not.

Delphi: `property HelpFile: string read FHelpFile write FHelpFile;`
C++: `__property AnsiString HelpFile = {read=FHelpFile, write=FHelpFile};`

- Specifies the name of the file the dialog uses to display Help. If no name is set, the Application.HelpFile is used.

Delphi: `property HelpType: THelpType read FHelpType write FHelpType;`
C++: `__property bool HelpType = {read=FHelpType, write=FHelpType, nodefault};`

- Indicates whether components invoke the help system using a context ID or a keyword

Delphi: `property HelpContext: THelpContext read FHelpContext write FHelpContext;`
C++: `__property bool HelpContext = {read=FHelpContext, write=FHelpContextp, nodefault};`
↳ Numeric ID for dalog's context-sensitive help topic, if HelpType 68 is set to *htContext*.

Delphi: `property HelpKeyword: string read FHelpKeyword write FHelpKeyword;`
C++: `__property AnsiString HelpKeyword = {read=FHelpKeyword, write=FHelpKeyword};`
↳ Keyword for dialog's context-sensitive help topic, if HelpType 68 is set to *htKeyword*.

Delphi: `property MaxFontSize: Integer read FMaxSize write FMaxSize;`
C++: `__property int MaxFontSize = {read=FMaxSize, write=FMaxSize, nodefault};`
↳ The maximum size of the font possible to set.

Delphi: `property MinFontSize: Integer read FMinSize write FMinSize;`
C++: `__property int MinFontSize = {read=FMinSize, write=FMinSize, nodefault};`
↳ The minimum size of the font possible to set.

Public Method

Delphi: `function Execute: Boolean;`
C++: `bool __fastcall Execute(void);`
↳ Execute opens the RtFont dialog, returning *true* when the user selects a font and clicks OK.

Delphi:

```
RtFontDialog1.Font := RtRichLabel1.Font;
if RtFontDialog1.Execute then
  RtRichLabel1.Font := RtFontDialog1.Font
```

C++:

```
RtFontDialog1->Font = RtRichLabel1->Font;
if (RtFontDialog1->Execute())
  RtRichLabel1->Font = RtFontDialog1->Font
```

Delphi: `property OnApply: TNotifyEvent read mApply write mApply;`
C++: `__property Classes::TNotifyEvent OnApply = {read=mApply, write=mApply, nodefault};`
↳ If you set the ShowApply 68 property to *true* the dialog will contain an Apply button. If the user clicks tis button this event will be triggered. This can be used to apply the current settings without closing the dialog.

3.2.4 TRtRichCaption Class

Unit/Namespace: `RtRichLabel`

Declarations

Delphi:
```
TRtRichCaption = type WideString;// previous to Studio 2009
TRtRichCaption = type UnicodeString;// for Studio 2009
```
C++:
```
typedef WideString TRtRichCaption;// previous to Studio 2009
typedef UnicodeString TRtRichCaption;// for Studio 2009
```

All captions used in the RtTools2D Cartesian graph controls use wide strings, in order to be able to display any Unicode character supplied by the font selected. The paint methods of the controls also provide support for special control sequences that result in enhanced formatting options 10.

Table: *Control sequences used in TRtRichCaption*

Sequence	Function
"\+"	switch superscript on/off
"\-"	switch subscript on/off
"\b"	switch bold style on/off
"\i"	switch italic style on/off
"\u"	switch underline style on/off
"\o"	switch strikeout style on/off
"\c[<RichId> 62]"	change color to ARGB value specified in brackets (red = '[A=255, R=255, G=0, B=0]', for example)
"\c[]"	change color back to ARGB value specified as initial foreground color
"\f[<Fontname>]"	change current font to the one specified by the font name in the brackets
"\f[]"	change current font back to the one specified as the initial font
"\\"	Insert "\" character

RtTools2D has a built-in system for retrieving variables from component properties by using their reference names. The captions and names of TRtGraph2D, TRtAxis and TRtSeries, as well as formulas of TRtFunctionSeries, can be inserted into any caption. If the string property changes, the related objects containing the TRtRichCaption properties will be updated automatically. The string variables are put into special brackets "<%" and "%>", and are named using their fully qualified names. For example, if you want to insert the ***Formula*** property of a TRtLinearRegression component named RegressionSeries, owned by a Form named ExampleForm, the code would be "<%ExampleForm.RegressionSeries.Formula%>".

You can insert results of numerical formulas into the caption in a similar way. Such a formula uses the syntax described in the overview 11 chapter and can contain numerical results of TRtFunctionSeries components. The output format can be included as string separated with a ";" sign, as with the ***FormatFloat*** function. As an example: you want to insert the relative error in percent of the slope of the above TRtLinearRegession component. The output should show 2 digits after the decimal separator. The code for this would be as follows: <% ExampleForm.RegressionSeries.DeltaSlope/ExampleForm.RegressionSeries. Slope*100;0.00%>.

3.2.5 TRtRichLabel Class

This unit defines some types used to control formatting and implements the label component 12 with enhanced formatting options and variable angle drawing.

Public Property

Delphi: `property PlainText: TRtRichCaption 70 read GetPlainText;`
C++: `__property TRtRichCaption PlainText = {read=GetPlainText};`
The caption 71 without any control sequences 70.

Published Properties

Delphi: `property Font: TRtFont 66 read GetFont write SetFont stored GetStoreFont;`
C++: `__property TRtFont* Font = {read=GetFont, write=SetFont, stored=GetStoreFont};`
The font with enhanced settings used to draw the caption.

Delphi: `property Angle: Single read FCaptionRec.Angle write SetAngle;`
C++: `__property float Angle = {read=FCaptionRec.ReadFontProperties, write=SetAngle};`
The angle in degrees at which the label should be drawn.

Delphi: `property BackColor: TRtColor 58 read FBackColor write SetBackColor;`
C++: `__property Rtgdi::TRtColor BackColor = {read=FBackColor, write=SetBackColor, nodefault};`
The background color of the control. Note that the control can be semi-transparent.

Delphi: `property ForeColor: TRtColor 58 read FForeColor write SetForeColor;`
C++: `__property Rtgdi::TRtColor ForeColor = {read=FForeColor, write=SetForeColor, nodefault};`
The text color of the control. Note that the text can be semi-transparent.

Delphi: `property Caption: TRtRichCaption 70 read GetCaption write SetCaption;`
C++: `__property WideString Caption = {read=GetCaption, write=SetCaption};`
The text of the control. Line breaks are allowed.

Delphi:
```
TRtContentAlignment = (TopLeft, TopCenter, TopRight,
MiddleLeft, MiddleCenter, MiddleRight, BottomLeft,
BottomCenter, BottomRight);
property TextAlign: TRtContentAlignment read GetAlign write
SetAlign;
```
C++:
```
enum TRtContentAlignment { TopLeft, TopCenter, TopRight,
MiddleLeft, MiddleCenter, MiddleRight, BottomLeft,
BottomCenter, BottomRight };
__property TRtContentAlignment TextAlign = {read=GetAlign,
write=SetAlign, nodefault};
```
The alignment of the text relative to the control dimensions rectangle.

3.2.6 TRtCaptionEdit Class

This class enables the editing of captions using enhanced formatting options.

Unit/Namespace: `RtCaptionEdit`

Declaration

Delphi:
```
TRtCaptionEdit = class(TCustomRichEdit); // VCL.net and
Studio 2009
TRtCaptionEdit = class(TTntCustomRichEdit); // Win32
```
C++:
```
class TRtCaptionEdit : public TTntCustomRichEdit;
```

Public Properties

Delphi:
```
property Bold: Boolean read GetBold write SetBold;
```
C++:
```
__property bool Bold = {read=GetBold, write=SetBold,
nodefault};
```
Retrieves and sets the bold style for the current selection. If nothing is selected, the next character entered will take the style.

Delphi:
```
property Italic: Boolean read GetItalic write SetItalic;
```
C++:
```
__property bool Italic = {read=GetItalic, write=SetItalic,
nodefault};
```
Retrieves and sets the italic style for the current selection. If nothing is selected, the next character entered will take the style.

Delphi:
```
property Underline: Boolean read GetUnderline write
SetUnderline;
```
C++:
```
__property bool Underline = {read=GetUnderline,
write=SetUnderline, nodefault};
```
Retrieves and sets the underline style for the current selection. If nothing is selected, the next character entered will take the style.

Delphi:
```
property Strikeout: Boolean read GetStrikeout write
SetStrikeout;
```
C++:
```
__property bool Strikeout = {read=GetStrikeout,
write=SetStrikeout, nodefault};
```
Retrieves and sets the strikeout style for the current selection. If nothing is selected, the next character entered will take the style.

Delphi:
```
property Superscript: Boolean read GetSuperscript write
SetSuperscript;
```
C++:
```
__property bool Superscript = {read=GetSuperscript,
write=SetSuperscript, nodefault};
```
Retrieves and sets the superscript style for the current selection. If nothing is selected, the next character entered will take the style.

Delphi:
```
property Subscript: Boolean read GetSubscript write
SetSubscript;
```
C++:
```
__property bool Subscript = {read=GetSubscript,
write=SetSubscript, nodefault};
```
Retrieves and sets the subscript style for the current selection. If nothing is selected, the next character entered will take the style.

Delphi: `property FontName: string read GetFontName write SetFontName;`
C++: `__property String FontName = {read=GetFontName, write=SetFontName};`
↳ Retrieves and sets the name of the font for the current selection. If nothing is selected, the next character entered will take the new font.

Delphi: `property SelColor: TColor read GetSelColor write SetSelColor;`
C++: `__property bool Graphics::TColor SelColor = {read=GetSelColor, write=SetSelColor, nodefault};`
↳ Retrieves and sets the color for the current selection. If nothing is selected, the next character entered will take the new color.

Delphi: `property SelVariable: string read GetSelVariable write SetSelVariable;`
C++: `__property String SelVariable = {read=GetSelVariable, write=SetSelVariable};`
↳ Retrieves and sets the variable or formula for the current selection.

Published Properties

Delphi: `property Caption: TRtRichCaption` 70 `read GetCaption write SetCaption`
C++: `__property TRtRichCaption Caption = {read=GetCaption, write=SetCaption};`
↳ Retrieves and sets the caption using enhanced formatting 10 options.

Delphi: `property Alignment: TAlignment read FAlignment write SetAlignment;`
C++: `__property Classes::TAlignment Alignment = {read=FAlignment, write=SetAlignment, nodefault};`
↳ Retrieves and sets the alignment (taLeftJustify, taRightJustify, taCenter) for the entire text.

Delphi: `property PropertyLinks: TRtPropertyLinks` 56 `read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`
↳ Links to update captions of other components to the Caption 73 property of the **TRtCaptionEdit** control.

3.2.7 TRtCheckBox Class

This component is almost an exact copy of the standard **TCheckBox**. Only the ***PropertyLinks*** property has been added.

Unit/Namespace: `RtStdLinked`

Declaration

Delphi: `TRtCheckBox = class(TCheckBox);`
C++: `class TRtCheckBox : public Stdctrls::TCheckBox;`

Public Property

Delphi: `property PropertyLinks: TRtPropertyLinks read FLink write FLink;` 56
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`

Links to update Boolean properties of other components to the ***Checked*** property of the **TRtCheckBox** control.

3.2.8 TRtRadioGroup Class

This component is almost an exact copy of the standard **TRadioGroup**. Only the ***PropertyLinks*** property has been added.

Unit/Namespace: `RtStdLinked`

Declaration

Delphi: `TRtCheckBox = class(TCheckBox);`
C++: `class TRtCheckBox : public Stdctrls::TCheckBox;`

Public Property

Delphi: `property PropertyLinks: TRtPropertyLinks read FLink write FLink;` 56
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`

Links to update integer properties of other components to the ***ItemIndex*** property of the **TRtRadioGroup** control.

3.2.9 TRtIntegerEdit Class

This class provides an edit control for entering integer numbers with range checking.

Unit/Namespace: `RtIntegerEdit`

Declaration

Delphi: `TRtIntegerEdit = class(TCustomEdit)`
C++: `class TRtIntegerEdit : public Stdctrls::TCustomEdit;`

Published Properties

Delphi: `property MinValue: Integer read FMinValue write SetMinValue;`
C++: `__property int MinValue = {read=FMinValue, write=SetMinValue, nodefault};`

- Minimum Value 75 that can be entered. If set equal to or greater than MaxValue 75, range checking will be switched off.

Delphi: `property MaxValue: Integer read FMaxValue write SetMaxValue;`
C++: `__property int MaxValue = {read=FMaxValue, write=SetMaxValue, nodefault};`

- Maximum Value 75 that can be entered. If set equal to or less than MinValue 75, range checking will be switched off.

Delphi: `property Value: Integer read FValue write SetValue;`
C++: `__property int Value = {read=FValue, write=SetValue, nodefault};`

- Value as **integer**. If the value changes OnValueChanged 75 will be triggered.

Delphi: `property PropertyLinks: TRtPropertyLinks` 56 `read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`

- Links to update integer properties of other components via the ***Value*** property of the **TRtIntegerEdit** control.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`

- Is triggered when the Value 75 has been changed.

3.2.10 TRtCustomDoubleEdit Class

This class is used as parent to the derived TRtDoubleEdit 77 and TRtNumericUpDn 78 classes. It provides an edit control for entering double or date/time numbers with range checking.

Unit/Namespace: `RtFloatEdit`

Declaration

Delphi: `TRtCustomDoubleEdit = class(TCustomEdit);`
C++: `class TRtCustomDoubleEdit : public Stdctrls::TCustomEdit;`

Published Properties

Delphi: `property MinValue: Double read FMinValue write SetMinValue;`
C++: `__property double MinValue = {read=FMinValue, write=SetMinValue, nodefault};`

- Minimum Value 76 that can be entered. If set equal to or greater MaxValue 76, range checking will be switched off.

Delphi: `property MaxValue: Double read FMaxValue write SetMaxValue;`
C++: `__property double MaxValue = {read=FMaxValue, write=SetMaxValue, nodefault};`

- Maximum Value 76 that can be entered. If set equal to or less than MinValue 76, range checking will be switched off.

Delphi: `property Value: Double read FValue write SetValue;`
C++: `__property double Value = {read=FValue, write=SetValue, nodefault};`

- Value as **double**. If the value changes OnValueChanged 76 will be triggered.

Delphi: `property PropertyLinks: TRtPropertyLinks 56 read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`

- Links to update floating point number properties of other components to the Value 76 property of the **TRtCustomDoubleEdit** control.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`

- Is triggered when the Value 76 changes.

3.2.11 TRtDoubleEdit Class

This class provides an edit control for entering double or date/time numbers with range checking.

Unit/Namespace: `RtFloatEdit`

Declaration

Delphi: `TRtDoubleEdit = class( TRtCustomDoubleEdit ;`
C++: `class TRtDoubleEdit : public TRtCustomDoubleEdit;`

Published Properties

Delphi: `property AsDateTime: Boolean read FAsDateTime write SetAsDateTime;`
C++: `__property bool AsDateTime = {read=FAsDateTime, write=SetAsDateTime, nodefault};`

If set to *false*, the Value 76 will be displayed as a floating point number using the defined Format 77. If set to *true*, it will be displayed as date/time value using the standard ***DateTimeToStr*** function.

Delphi: `property AsTimeSpan: Boolean read FAsTimeSpan write SetAsTimeSpan;`
C++: `__property bool AsTimeSpan = {read=FAsTimeSpan, write=SetAs TimeSpan, nodefault};`

If set to *false*, the Value 76 will be displayed as a floating point number using the defined Format 77. If set to *true*, it will be displayed as date/time span value displaying days.hours:minutes:seconds.milliseconds between stop and start time. This option can be used to enter MovingSlitWidth 145 and MovingSlitIncrement 145 for axis with scaling set to *DateTime*.

Delphi: `property DoHashes: Boolean read FDoHashes write FDoHashes;`
C++: `__property bool DoHashes = {read=FDoHashes, write=FDoHashes, nodefault};`

If set to *false*, the Value 76 will be displayed normally. If set to *true* and the number characters would exceed the width of the control "#" characters will be displayed instead.

Delphi: `property Format: string read FFormat write SetFormat;`
C++: `__property AnsiString Format = {read=FFormat, write=SetFormat};`

If AsDateTime 77 was set to *false*, you can specify a format string to display the number when the control is not focused. This string uses the same syntax as the standard ***FormatFloat*** function. If the format string is empty, then all significant digits of the number will be output.

3.2.12 TRtNumericUpDn Class

This class provides an edit control with up and down buttons for incrementing and decrementing values.

Unit/Namespace: `RtFloatEdit`

Declaration

Delphi: `TRtNumericUpDn = class(TRtCustomDoubleEdit);`
C++: `class TRtNumericUpDn : public TRtCustomDoubleEdit;`

Published Properties

Delphi: `property AsInteger: Integer read GetAsInteger write SetAsInteger;`
C++: `__property int AsInteger = {read=GetAsInteger, write=SetAsInteger, nodefault};`

Reads and writes the value as an **integer**.

Delphi: `property Increment: Double read FIncrement write FIncrement;`
C++: `__property double Increment = {read=FIncrement, write=FIncrement};`

The value to be added or subtracted when one of the up or down buttons is pressed.

Delphi: `property BtnWidth: Integer read FBtnWidth write SetBtnWidth;`
C++: `__property int BtnWidth = {read=FBtnWidth, write=SetBtnWidth, nodefault};`

The size of the up and down buttons at the right of the control.

Delphi: `property DecimalPlaces: Integer read FDigits write SetDigits;`
C++: `__property int DecimalPlaces = {read=FDigits, write=SetDigits, nodefault};`

The number is displayed with a fixed decimal separator. This property defines the amount of digits to be displayed after the separator. Setting this to 0 will display the number as an integer.

3.3 Style Selection Controls

3.3.1 TRtColorPickCombo Class

This class provides a control to select colors from a palette or a color settings tool window.

Unit/Namespace: `RtColorPicker`

Declaration

Delphi: `TRtColorPickCombo = class(TCustomComboBox);`
C++: `class TRtColorPickCombo : public Stdctrls::TCustomComboBox;`

Public Property

Delphi: `property AsTColor: TColor read GetAsTColor write SetAsTColor;`
C++: `__property Graphics::TColor AsTColor = {read=GetAsTColor, write=SetAsTColor, nodefault};`

↳ Read or write the ActiveColor 79 as a **TColor** value. Set EnableAlpha 79 to *false* if you are using the control to set standard **TColor** values.

Published Properties

Delphi: `property ActiveColor: TRtColor 58 read FActiveColor write SetActiveColor;`
C++: `__property Rtgdi::TRtColor ActiveColor = {read=FActiveColor, write=SetActiveColor, nodefault};`

↳ The currently selected color.

Delphi: `property EnableAlpha: Boolean read FEnableAlpha write FEnableAlpha;`
C++: `__property bool EnableAlpha = {read=FEnableAlpha, write=FEnableAlpha, nodefault};`

↳ If set to *true*, the control will support transparent colors.

Delphi: `property DirectlyShowTuneColors: Boolean read FDirectlyShowTuneColors write FDirectlyShowTuneColors;`
C++: `__property bool DirectlyShowTuneColors = {read=FDirectlyShowTuneColors, write=FDirectlyShowTuneColors, nodefault};`

↳ If set to *true*, the control will not provide a palette dropdown, but will show Tune Colors Dialog 15 directly instead.

Delphi: `property DialogTitle: string read FDialogTitle write FDialogTitle;`
C++: `__property AnsiString DialogTitle = {read=FDialogTitle, write=FDialogTitle};`

↳ The title shown in the Tune Colors Dialog 15.

Delphi: `property OtherBtnCaption: TCaption read FOtherBtnCaption write FOtherBtnCaption;`
C++: `__property AnsiString OtherBtnCaption = {read=FOtherBtnCaption, write=FOtherBtnCaption};`

↳ The caption on the button that opens the Tune Colors Dialog 15.

Delphi: `property PropertyLinks: TRtPropertyLinks 56 read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`

↳ Links to update color properties of other components to the ActiveColor 79 property of the **TRtColorPickCombo** control.

Event

Delphi: `property OnChanged: TNotifyEvent read FOnChanged write FOnChanged;`
C++: `__property Classes::TNotifyEvent OnChanged = {read=FOnChanged, write=FOnChanged};`

↳ Event triggered when ActiveColor 79 changes.

3.3.2 TRtTuneColorsDialog Class

This class provides a replacement for the standard Windows color dialog. It also supports transparency.

Unit/Namespace: `RtTuneColorsDialog`

Declaration

Delphi: `TRtTuneColorsDialog = class(TComponent);`
C++: `class TRtTuneColorsDialog : public Classes::TComponent;`

Public Property

Delphi: `property AsTColor: TColor read GetAsTColor write SetAsTColor;`
C++: `__property Graphics::TColor AsTColor = {read=GetAsTColor, write=SetAsTColor, nodefault};`

↳ Read or write the ActiveColor 80 as a **TColor** value. Set EnableAlpha 80 to *false* if you are using the control to set standard **TColor** values.

Published Properties

Delphi: `property ActiveColor: TRtColor 58 read FActiveColor write SetActiveColor;`
C++: `__property Rtgdi::TRtColor ActiveColor = {read=FActiveColor, write=SetActiveColor, nodefault};`

↳ The currently selected color.

Delphi: `property EnableAlpha: Boolean read FEnableAlpha write FEnableAlpha;`
C++: `__property bool EnableAlpha = {read=FEnableAlpha, write=FEnableAlpha, nodefault};`

↳ If set to *true*, the control will support transparent colors.

Delphi: `property Title: string read FTitle write FTitle;`
C++: `__property AnsiString Title = {read=FTitle, write=FTitle};`

↳ The title shown in the dialog.

Delphi: `property PropertyLinks: TRtPropertyLinks 56 read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`

↳ Links to update color properties of other components to the ***ActiveColor*** property of the **TRtColorPickCombo** control.

Public Method

Delphi: `function Execute: Boolean;`
C++: `bool __fastcall Execute(void);`

↳ Shows the dialog in a modal window. If the user clicks OK, the dialog returns *true*. If the user clicks cancel or closes the window, *false* is returned.

3.3.3 TRtDashStylesList Class

This class provides a list box for selecting line dash styles. It can be used to set up series line styles.

Unit/Namespace: `RtDashStylesList`

Declaration

Delphi: `TRtDashStylesList = class(TCustomListBox);`
C++: `class TRtDashStylesList : public Stdctrls::TCustomListBox;`

Published Properties

Delphi: `property DashStyle: TRtDashStyle` 58 `read GetDashStyle write SetDashStyle;`
C++: `__property Gdipapi::TDashStyle DashStyle = {read=GetDashStyle, write=SetDashStyle, default=0};`
The currently selected line dash style.

Delphi: `property LineColor: TRtColor` 58 `read FLineColor write SetLineColor;`
C++: `__property Rtgdi::TRtColor LineColor = {read=FLineColor, write=SetLineColor, nodefault};`
The color of the line dash style samples drawn in the selection list.

Delphi: `property LineWidth: Single read FLineWidth write SetLineWidth;`
C++: `__property float LineWidth = {read=FLineWidth, write=SetLineWidth};`
The width of the line dash style samples drawn in the selection list.

Delphi: `property PropertyLinks: TRtPropertyLinks` 56 `read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`
Links to update dash style properties of other components to the DashStyle 81 property of the **TRtDashStylesList** control.

Public Property

Delphi: `property UseLineSettings: TRtSimpleLineSettings` 99 `read FUseLineSettings write SetUseLineSettings;`
C++: `__property Rtstyles::TRtSimpleLineSettings* UseLineSettings = {read=FUseLineSettings, write=SetUseLineSettings};`
Keep this property *nil* if you want the control to draw the line samples using LineColor 81 and LineWidth 81. If set to a line settings class e.g. from a series, a graph grid or a legend frame. The sample lines will be drawn using these settings. This will allow to draw gradient pen lines.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`
Event triggered when DashStyle 81 changes.

3.3.4 TRtDashStylesCombo

This class provides a combo box that provides a dropdown list for selecting line dash styles. It can be used to set up series line styles.

Unit/Namespace: `RtDashStylesCombo`

Declaration

Delphi: `TRtDashStylesCombo = class(TCustomComboBox);`
C++: `class TRtDashStylesCombo : public Stdctrls::TCustomComboBox`

Published Properties

Delphi: `property DashStyle: TRtDashStyle` 58 `read GetDashStyle write SetDashStyle;`
C++: `__property Gdipapi::TDashStyle DashStyle = {read=GetDashStyle, write=SetDashStyle, default=0};`
The currently selected line dash style.

Delphi: `property LineColor: TRtColor` 58 `read FLineColor write SetLineColor;`
C++: `__property Rtgdi::TRtColor LineColor = {read=FLineColor, write=SetLineColor, nodefault};`
The color of the line dash style samples drawn in the selection list.

Delphi: `property LineWidth: Single read FLineWidth write SetLineWidth;`
C++: `__property float LineWidth = {read=FLineWidth, write=SetLineWidth};`
The width of the line dash style samples drawn in the selection list.

Delphi: `property PropertyLinks: TRtPropertyLinks` 56 `read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`
Links to update dash style properties of other components to the DashStyle 82 property of the **TRtDashStylesCombo** control.

Delphi: `property UseLineSettings: TRtSimpleLineSettings` 99 `read FUseLineSettings write SetUseLineSettings;`
C++: `__property Rtstyles::TRtSimpleLineSettings* UseLineSettings = {read=FUseLineSettings, write=SetUseLineSettings};`
Keep this property *nil* if you want the control to draw the line samples using LineColor 81 and LineWidth 81. If set to a line settings class e.g. from a series, a graph grid or a legend frame. The sample lines will be drawn using these settings. This will allow to draw gradient pen lines.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`
Event triggered when DashStyle 82 changes.

3.3.5 TRtAreaStylesList Class

This class provides a list box for selecting hatched area fill styles. It can be used to set up series area styles.

Unit/Namespace: `RtAreaStylesList`

Declaration

Delphi: `TRtAreaStylesList = class(TCustomListBox);`
C++: `class TRtAreaStylesList : public Stdctrls::TCustomListBox;`

Published Properties

Delphi: `property AreaStyle: TRtAreaStyle read GetAreaStyle write SetAreaStyle;`
C++: `__property Rtgdi::TRtAreaStyle AreaStyle = {read=GetAreaStyle, write=SetAreaStyle, nodefault};`
The currently selected area hatch fill style.

Delphi: `property AreaForeColor: TRtColor read FAreaForeColor write SetAreaForeColor;`
C++: `__property Rtgdi::TRtColor AreaForeColor = {read=FAreaForeColor, write=SetAreaForeColor, nodefault};`
The foreground drawing color for the hatch fill style sample items.

Delphi: `property AreaBackColor: TRtColor read FAreaBackColor write SetAreaBackColor;`
C++: `__property Rtgdi::TRtColor AreaBackColor = {read=FAreaBackColor, write=SetAreaBackColor, nodefault};`
The background drawing color for the hatch fill style sample items.

Delphi: `property PropertyLinks: TRtPropertyLinks read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`
Links to update area style properties of other components to the AreaStyle property of the **TRtAreaStylesList** control.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`
Event triggered when AreaStyle changes.

3.3.6 TRtAreaStylesCombo Class

This class provides a combo box that provides a dropdown list for selecting hatched area fill styles. It can be used to set up series area styles.

Unit/Namespace: `RtAreaStylesCombo`

Declaration

Delphi: `TRtAreaStylesCombo = class(TCustomComboBox);`
C++: `class TRtAreaStylesCombo: public Stdctrls::TCustomComboBox;`

Published Properties

Delphi: `property AreaStyle: TRtAreaStyle` 59 `read GetAreaStyle write SetAreaStyle;`
C++: `__property Rtgdi::TRtAreaStyle AreaStyle = {read=GetAreaStyle, write=SetAreaStyle, nodefault};`
The currently selected area hatch fill style.

Delphi: `property ShowPickList: Boolean read FDropPickList write SetDropPickList;`
C++: `__property bool ShowPickList = {read=FDropPickList, write=SetDropPickList, nodefault};`
If set *true*, the control drops down a multi column list for selection else a single column standard drop down selection.

Delphi: `property AreaForeColor: TRtColor` 58 `read FAreaForeColor write SetAreaForeColor;`
C++: `__property Rtgdi::TRtColor AreaForeColor = {read=FAreaForeColor, write=SetAreaForeColor, nodefault};`
The foreground drawing color for the hatch fill style sample items.

Delphi: `property AreaBackColor: TRtColor` 58 `read FAreaBackColor write SetAreaBackColor;`
C++: `__property Rtgdi::TRtColor AreaBackColor = {read=FAreaBackColor, write=SetAreaBackColor, nodefault};`
The background drawing color for the hatch fill style sample items.

Delphi: `property PropertyLinks: TRtPropertyLinks` 56 `read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`
Links to update area style properties of other components to the AreaStyle 84 property of the **TRtAreaStylesCombo** control.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`
Event triggered when AreaStyle 84 changes.

3.3.7 TRtPointSymbolsList Class

This class provides a list box for selecting point symbol styles. It can be used to set up series point styles.

Unit/Namespace: `RtPointSymbolsList`

Declaration

Delphi: `TRtPointSymbolsList = class(TCustomListBox);`
C++: `class TRtPointSymbolsList : public Stdctrls::TCustomListBox;`

Published Properties

Delphi: `property Symbol: TRtPointSymbol` 63 `read GetPointSymbol write SetPointSymbol;`
C++: `__property Rtdrawing::TRtPointSymbol Symbol = {read=GetPointSymbol, write=SetPointSymbol, nodefault};`
The currently selected point symbol drawing type.

Delphi: `property SymbolBorderColor: TRtColor` 58 `read FSymbolBorderColor write SetSymbolBorderColor;`
C++: `__property Rtgdi::TRtColor SymbolBorderColor = {read=FSymbolBorderColor, write=SetSymbolBorderColor, nodefault};`
The color used to draw the border line of the symbols example list.

Delphi: `property SymbolFillColor: TRtColor` 58 `read FSymbolFillColor write SetSymbolFillColor;`
C++: `__property Rtgdi::TRtColor SymbolFillColor = {read=FSymbolFillColor, write=SetSymbolFillColor, nodefault};`
The color used for the inner fill of the symbols example list.

Delphi: `property SymbolBorderLineWidth: Single read FSymbolBorderLineWidth write SetSymbolBorderLineWidth;`
C++: `__property float SymbolBorderLineWidth = {read=FSymbolBorderLineWidth, write=SetSymbolBorderLineWidth};`
The line width used when drawing the border of the symbols in the example list.

Delphi: `property PropertyLinks: TRtPropertyLinks` 56 `read FLink write FLink;`
C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`
Links to update point symbol properties of other components with the Symbol 85 property of the **TRtPointSymbolsList** control.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`
Event triggered when Symbol 85 changes.

3.3.8 TRtPointSymbolsCombo Class

This class provides a combo box dropping down a list for selecting point symbol styles. It can be used to set up series point styles.

Unit/Namespace: `RtPointSymbolsCombo`

Declaration

Delphi: `TRtPointSymbolsCombo = class(TCustomComboBox);`

C++:
```
class TRtPointSymbolsCombo : public Stdctrls::
TCustomComboBox;
```

Published Properties

Delphi: `property Symbol: TRtPointSymbol` 63 `read GetPointSymbol write SetPointSymbol;`

C++: `__property Rtdrawing::TRtPointSymbol Symbol = {read=GetPointSymbol, write=SetPointSymbol, nodefault};`

The currently selected point symbol drawing type.

Delphi: `property ShowPickList: Boolean read FDropPickList write SetDropPickList;`

C++: `__property bool ShowPickList = {read=FDropPickList, write=SetDropPickList, nodefault};`

If set *true*, the control provides a multi-column dropdown list for selection; otherwise, a standard single-column dropdown selection list is provided.

Delphi: `property SymbolBorderColor: TRtColor` 58 `read FSymbolBorderColor write SetSymbolBorderColor;`

C++: `__property Rtgdi::TRtColor SymbolBorderColor = {read=FSymbolBorderColor, write=SetSymbolBorderColor, nodefault};`

The color used for the border line of the symbols example list.

Delphi: `property SymbolFillColor: TRtColor` 58 `read FSymbolFillColor write SetSymbolFillColor;`

C++: `__property Rtgdi::TRtColor SymbolFillColor = {read=FSymbolFillColor, write=SetSymbolFillColor, nodefault};`

The color used for the the inner fill of the symbols example list.

Delphi: `property SymbolBorderLineWidth: Single read FSymbolBorderLineWidth write SetSymbolBorderLineWidth;`

C++: `__property float SymbolBorderLineWidth = {read=FSymbolBorderLineWidth, write=SetSymbolBorderLineWidth};`

The line width used when drawing the border of the symbols example list.

Delphi: `property PropertyLinks: TRtPropertyLinks` 56 `read FLink write FLink;`

C++: `__property Rtpropertylink::TRtPropertyLinks* PropertyLinks = {read=FLink, write=FLink};`

Links to update point symbol properties of other components with the Symbol 86 property of the **TRtPointSymbolsCombo** control.

Event

Delphi: `property OnValueChanged: TNotifyEvent read mValueChanged write mValueChanged;`
C++: `__property Classes::TNotifyEvent OnValueChanged = {read=mValueChanged, write=mValueChanged};`
↳ Event triggered when Symbol 86 changes.

3.4 Undo Support

3.4.1 TRtUndoStack Class

This class provides a component that implements undo/redo functionality.

Unit/Namespace: `RtUndoStack`

Declaration

Delphi: `TRtUndoStack = class(TComponent);`
C++: `class TRtUndoStack : public Classes::TComponent`

Public Properties

Delphi: `property Restoring: Boolean read FRestoring;`
C++: `__property bool Restoring = {read=FRestoring, nodefault}`
↳ During the undo/redo process this property returns *true*.

Delphi: `property CanUndo: Boolean read GetCanUndo;`
C++: `__property bool CanUndo = {read=GetCanUndo, nodefault};`
↳ This property returns *true* if something can be undone, i.e. the undo stack contains one or more items. You can use this property to enable/disable undo tool buttons or actions in an ***OnUpdate*** event.

Delphi: `property CanRedo: Boolean read GetCanRedo;`
C++: `__property bool CanRedo = {read=GetCanRedo, nodefault};`
↳ This property returns *true* if something can be redone, i.e. the redo stack contains one or more items. You can use this property to enable/disable undo tool buttons or actions in an ***OnUpdate*** event.

Delphi: `property UndoCount: Integer read GetUndoCount;`
C++: `__property int UndoCount = {read=GetUndoCount, nodefault};`
↳ The number of items on the undo stack.

Delphi: `property RedoCount: Integer read GetRedoCount;`
C++: `__property int RedoCount = {read=GetRedoCount, nodefault};`
↳ The number of items on the redo stack.

Delphi: `property UndoList: TStrings read FUndoList;`
C++: `__property Classes::TStrings* UndoList = {read=FUndoList};`
↳ The list of all comments passed to the ***Push*** 88 methods or ***StartGroup*** 90. Used internally to display the dropdown list for the **TRtUndoButton** 91 component.

Delphi: `property RedoList: TStrings read FUndoList;`
C++: `__property Classes::TStrings* UndoList = {read=FUndoList};`

- The list of all comments passed to the ***Push*** 88 methods or ***StartGroup*** 90. Used internally to display the dropdown list for the **TRtRedoButton** 92 component.

Published Properties

Delphi: `property Enabled: Boolean read FEnabled write FEnabled;`
C++: `__property bool Enabled = {read=FEnabled, write=FEnabled, nodefault};`

- If this property is set to *false*, the component will not store anything onto the stack. Useful for temporarily disabling auto-storage of graph control property changes.

Delphi: `property Limit: Integer read FLimit write FLimit;`
C++: `__property int Limit = {read=FLimit, write=FLimit, nodefault};`

- Maximum depth of the stack for storing component changes. Set to 0 to enable an infinite depth.

Delphi: `property ToMemoryStream: Boolean read FToMemoryStream write FToMemoryStream;`
C++: `__property bool ToMemoryStream = {read=FToMemoryStream, write=FToMemoryStream, nodefault};`

- The items can be stored either to temporary files (*false*) or to a RAM memory stream (*true*).

Events

Delphi: `property OnUndo: TNotifyEvent read mUndo write mUndo;`
C++: `__property Classes::TNotifyEvent OnUndo = {read=mUndo, write=mUndo};`

- This event is triggered at the end of the undo method. It can be used to update the display of any dependent controls.

Delphi: `property OnRedo: TNotifyEvent read mRedo write mRedo;`
C++: `__property Classes::TNotifyEvent OnRedo = {read=mRedo, write=mRedo};`

- This event is triggered at the end of the redo method. It can be used to update the display of any dependent controls.

Public Methods

Delphi: `procedure PushProperty(Instance: TObject; const Comment, PropName: string);`
C++: `void __fastcall PushProperty(System::TObject* Instance, const AnsiString Comment, const AnsiString PropName);`

- This method is the core of the undo stack component. It can used to store the state of almost any published property of any object in your program. The ***Instance*** parameter takes as its value the object you want to alter a property for afterwards, while the ***Comment*** parameter takes as its value a description of the operation that can then be displayed with the **TRtUndoButton** 91 and **TRtRedoButton** 92 controls. The ***PropName*** parameter takes the name of the property to be stored. Here, as an example, is the code used internally to change the visibility of a line:

Delphi:

```
procedure TRtLineSettings.SetVisible(Value: Boolean);
// sets the visibility of the line
begin
  if FVisible<>Value then
  begin
    if Assigned(FUndoStack) then
      FUndoStack.PushProperty(Self,'Line visible','Visible');
    FVisible := Value;
    Modified;
  end;
end;
```

C++:

```
void __fastcall TRtLineSettings::SetVisible(bool Value)
// sets the visibility of the line
{
   if (FVisible != Value)
   {
      if (FUndoStack != null)
      {
         FUndoStack->PushProperty(this,"Line visible","Visible")
      }
      FVisible = Value;
      Modified();
   }
}
```

Property types supported are: any ordinal type as integer, Boolean, char, wchar, enumerations, sets, any floating point type, strings and wide strings, one dimensional dynamic arrays of the above simple types and any type storable as a variant.

Delphi: `procedure PushComponent(AComponent: TComponent; const Comment: string);`

C++: `void __fastcall PushComponent(Classes::TComponent* AComponent, const AnsiString Comment);`

↳ Stores all the ***AComponents*** properties onto the stack. Uses the internal streaming capabilities of the component. Can be used to store a component before deleting it, enabling deletion to be undone.

Delphi: `procedure PushFreeComponent(AComponent: TComponent; const Comment: string);`

C++: `void __fastcall PushFreeComponent(Classes::TComponent* AComponent, const AnsiString Comment);`

↳ Stores the reference to the component onto the stack, indicating it should be deleted on undo. This method can be used on creation of an object. The following example is taken from the component editor of the graph component, which can add new data vectors:

Delphi:

```
procedure TDsgnGraphEdit.AddVectorButtonClick(Sender: TObject);
var dVec: TRtDoubleVector;
begin
  dVec := TRtDoubleVector(FormsDesigner.CreateComponent(TRtDoubleVector,Graph.Owner,0,0,0,
0));
  if Assigned(FUndoStack) then
    FUndoStack.PushFreeComponent(dVec,'New double vector');
  UpdateCombos;
end;
```

Delphi: `procedure PushEditState(AEdit: TCustomEdit; ACaption: TRtRichCaption; ASelStart,ASelLength: Integer; Comment: string);`
C++: `void __fastcall PushEditState(Stdctrls::TCustomEdit* AEdit, TRtRichCaption ACaption, int ASelStart, int ASelLength, AnsiString Comment);`

This method is specialized for the storage of the state of any edit control. It stores the caption text and the current selection.

Delphi: `procedure PushVectorState(AVector: TRtDoubleVector; const Comment: string);`
C++: `void __fastcall PushVectorState(Rtvectors::TRtDoubleVector* AVector, const AnsiString Comment);`

This method stores all data contained in a **TRtDoubleVector** component.

Delphi: `procedure StartGroup(const Comment: string);`
C++: `void __fastcall StartGroup(const AnsiString Comment);`

Indicates that multiple subsequent operations are to be stored/restored together, until ***StopGroup***. Please always ensure you use within StartGroup/StopGroup pairs. Groups can be nested.

Delphi:

```
procedure TRtAxis.SetCaptionFont(Value: TFont);
begin
  if Assigned(FUndoStack) then
  begin
    FUndoStack.StartGroup('Font of axis caption');
    FUndoStack.PushProperty(FCaptionRec.Font,'','Name');
    FUndoStack.PushProperty(FCaptionRec.Font,'','Style');
    FUndoStack.PushProperty(FCaptionRec.Font,'','Size');
    FUndoStack.StopGroup;
  end;
  FCaptionRec.Font.Assign(Value);
end;
```

C++:

```
void __fastcall TRtAxis::SetCaptionFont(TFont Value)
// sets the visibility of the line
{
   if (FUndoStack != null)
   {
      FUndoStack->StartGroup("Font of axis caption");
      FUndoStack->PushProperty(FCaptionRec::Font,"","Name");
      FUndoStack->PushProperty(FCaptionRec::Font,"","Style");
      FUndoStack->PushProperty(FCaptionRec::Font,"","Size");
      FUndoStack->StopGroup();
   }
   FCaptionRec::Font->Assign(Value);
}
```

Delphi: `procedure StopGroup;`
C++: `void __fastcall StopGroup(void);`

Indicates that multiple prior operations have been stored/restored together via ***StartGroup***. Please always ensure you use within StartGroup/StopGroup pairs. Groups can be nested.

Delphi: `procedure Undo(ToIndex: Integer = 0);`
C++: `void __fastcall Undo(int ToIndex = 0x0);`

Call without any parameter to undo the last operation, or include the index to restore up to the parameter given.

Delphi: `procedure Redo(ToIndex: Integer = 0);`
C++: `void __fastcall Redo(int ToIndex = 0x0);`
↳ Call without any parameter to redo the last operation, or include the index to replay up to the parameter given.

Delphi: `procedure Clear;`
C++: `void __fastcall Clear(void);`
↳ Call to clear the stack.

3.4.2 TRtDropDnButton Class

This class provides a button control that is specialized for the navigation in the **TRtUndoStack** 87 component and thus very similar to **TSpeedButton**. It supplies the basic functionality for the **TRtUndoButton** 92 and **TRtRedoButton** 92 controls. Most of the properties are identical with those of the **TSpeedbutton** control.

Unit/Namespace: `RtUndoButtons`

Declaration

Delphi: `TRtDropDnButton = class(TGraphicControl);`
C++: `class TRtDropDnButton : public Controls::TGraphicControl;`

Published Properties

Delphi: `property UndoStack: TRtUndoStack` 87 `read FUndoStack write FUndoStack;`
C++: `__property Rtundostack::TRtUndoStack* UndoStack = {read=FUndoStack, write=FUndoStack};`
↳ The reference to the undo stack object which you want to navigate.

Delphi: `property Glyph: TBitmap read FGlyph write SetGlyph;`
C++: `__property Graphics::TBitmap* Glyph = {read=FGlyph, write=SetGlyph};`
↳ The bitmap object containing the image that appears on the face of the button. Depending on the ***NumGlyphs*** 91 property, ***Glyph*** can provide up to four images within a single bitmap, corresponding to the unselected, disabled, highlighted and down state. If ***NumGlyphs*** is set to *1*, then the other states are generated from the first picture.

Delphi: `property NumGlyphs: TNumGlyphs read FNumGlyphs write SetNumGlyphs default 2;`
C++: `__property Buttons::TNumGlyphs NumGlyphs = {read=FNumGlyphs, write=SetNumGlyphs, default=2};`
↳ The number of images provided by the bitmap assigned to the ***Glyph*** 91 property.

Delphi: `property Layout: TButtonLayout read FLayout write SetLayout default blGlyphLeft;`
C++: `__property Buttons::TButtonLayout Layout = {read=FLayout, write=SetLayout, default=0};`
↳ The relationship of the positions of the ***Glyph*** and the ***Caption*** of the button: image at left, right, top and bottom.

Delphi: `property Margin: Integer read FMargin write SetMargin default -1;`
C++: `__property int Margin = {read=FMargin, write=SetMargin, default=-1};`

The indentation of the image specified by the ***Glyph*** or the ***Caption***. If set to –1, the image or text is centered on the button.

Delphi: `property Spacing: Integer read FSpacing write SetSpacing default 4;`
C++: `__property int Spacing = {read=FSpacing, write=SetSpacing, default=4};`

The spacing in pixels between the ***Glyph*** and the ***Caption***.

Delphi: `property Transparent: Boolean read FTransparent write SetTransparent default True;`
C++: `__property bool Transparent = {read=FTransparent, write=SetTransparent, default=1};`

Specifies whether the background of the button is transparent.

Delphi: `property Flat: Boolean read FFlat write SetFlat default True;`
C++: `__property bool Flat = {read=FFlat, write=SetFlat, default=1};`

If set *true*, the button will only show raised borders when the mouse is over the control.

3.4.3 TRtUndoButton Class

This class provides a button control that is specialized for the navigation in the undo list of the **TRtUndoStack** 87 component and thus very similar to **TSpeedButton**.

Unit/Namespace: `RtUndoButtons`

Declaration

Delphi: `TRtUndoButton = class(TRtDropDnButton);` 91
C++: `class TRtUndoButton : public TRtDropDnButton;`

3.4.4 TRtRedoButton Class

This class provides a button control that is specialized for the navigation in the redo list of the **TRtUndoStack** 87 component and thus very similar to **TSpeedButton**.

Unit/Namespace: `RtUndoButtons`

Declaration

Delphi: `TRtRedoButton = class(TRtDropDnButton);` 91
C++: `class TRtRedoButton : public TRtDropDnButton;`

3.5 Cartesian Plot Components

3.5.1 TRtPointVector Class

This class provides a non-visual component for the storage of TRtPoint 59 values in an array that is automatically adjusted for size. The data can be accessed via the default property Items 93 as it is an array[0..MaxInteger] of TRtPoint 59. It is used to store the internal graph point data of the series.

Unit/Namespace: `RtVectors`

Declaration

Delphi: `TRtPointVector = class(TComponent);`
C++: `class TRtPointVector : public System::TObject;`

Published Properties

Delphi: `property Items[i: Integer]: TRtPoint read GetValue write SetValue; default;`
C++: `__property Gdipapi::TGPPointF Items[int i] = {read=GetValue, write=SetValue/*, default*/};`
↳ The default indexed property gives access to the internal dynamically-sized storage array for the points.

Delphi: `property Count: Integer read GetCount;`
C++: `__property int Count = {read=FCount, nodefault};`
↳ The count of the points stored.

Delphi: `property Capacity: Integer read FCapacity write SetCapacity;`
C++: `__property int Capacity = {read=FCapacity, write=SetCapacity, nodefault};`
↳ Size of the array allocated.

Delphi: `property X[i: Integer]: Single read GetX write SetX;`
C++: `__property float X[int i] = {read=GetX, write=SetX};`
↳ Indexed property for accessing the X-component of the internal dynamically-sized storage array for the points.

Delphi: `property Y[i: Integer]: Single read GetY write SetY;`
C++: `__property float Y[int i] = {read=GetY, write=SetY};`
↳ Indexed property for accessing the Y-component of the internal dynamically-sized storage array for the points.

Delphi: `property Values: TRtPointArray 59 read FArray;`
C++: `__property Gdipapi::TPointFDynArray Values = {read=FArray};`
↳ The dynamically-sized storage array for the points.

Public Methods

Delphi: `function Add(const n: TRtPoint 59): Integer;`
C++: `int __fastcall Add(const Gdipapi::TGPPointF &n);`
↳ Extends the storage array, giving it the capacity 93 for storing one more point. Sets the last item to the new value ***n***. Returns the new count 93.

Delphi: `procedure Clear;`
C++: `void __fastcall Clear(void);`
↳ Clears the points. Sets count 93 to 0.

3.5.2 TRtDoubleVector Class

1.234 6.789 45.67 This class provides a non-visual component for the storage of **double** or **TDateTime** values in an array that is automatically adjusted for size. The data can be accessed via the default property Items 94 as it is an array[0..MaxInteger] of doubles. It can be also used as a link to a database holding the data. This class needs to be assigned as graph series 26 X/Y point values.

Unit/Namespace: `RtVectors`

Declaration

Delphi:
```
TRtustomDoubleVector = class(TComponent);
TRtDoubleVector = class(TRtCustomDoubleVector);
```
C++:
```
class TRtCustomDoubleVector : public Classes::TComponent;
class TRtDoubleVector : public TRtCustomDoubleVector;
```

Public Properties

Delphi:
```
property Items[i:Integer]:Double read GetValue write
SetValue; default;
```
C++:
```
__property double Items[int i] = {read=GetValue,
write=SetValue/*, default*/};
```
↳ The default indexed property gives access to the internal dynamically-sized storage array of the **double** values.

Delphi: `property Count: Integer read GetCount;`
C++: `__property int Capacity = {read=FCapacity, write=SetCapacity, nodefault};`
↳ The count of the data stored.

Delphi: `property Minimum: Double read GetMinimum;`
C++: `__property double Minimum = {read=GetMinimum};`
↳ The minimum value of the data contained in the internal array, excluding values marked as outliers 95.

Delphi: `property Maximum: Double read GetMaximum;`
C++: `__property double Maximum = {read=GetMaximum};`
↳ The maximum value of the data contained in the internal array, excluding values marked as outliers 95.

Delphi: `property MinimumIdx: Integer read GetMinimumIdx;`
C++: `__property int MinimumIdx = {read=GetMinimumIdx, nodefault};`
↳ The index of the minimum value of the data contained in the internal array, excluding values marked as outliers 95.

Delphi: `property MaximumIdx: Integer read GetMaximumIdx;`
C++: `__property int MaximumIdx = {read=GetMaximumIdx, nodefault};`
↳ The index of the maximum value of the data contained in the internal array, excluding values marked as outliers 95.

Delphi: `property Sum: Double read GetSum;`
C++: `__property double Sum = {read=GetSum};`
↳ Returns the sum of all numbers contained in the internal array, excluding values marked as outliers 95.

Delphi: `property OnlyAscending: Boolean read GetOnlyAscending;`
C++: `__property bool OnlyAscending = {read=GetOnlyAscending, nodefault};`
↳ Returns *true* if all values are sorted in ascending order. Used for the acceleration of internal calculations.

Delphi: `property AnyOutliers: Boolean read FAnyOutliers;`
C++: `__property bool AnyOutliers = {read=FAnyOutliers, nodefault};`
↳ Indicates if any item has been marked as an outlier.

Delphi: `property Outliers[i:Integer]: Boolean read IsOutlier write SetOutlier;`
C++: `__property bool Outliers[int i] = {read=IsOutlier, write=SetOutlier};`
↳ This indexed property accesses the outliers as an array of Boolean values.

Delphi: `property MinimumIncludingOutliers: Double read GetMinimumIncludingOutliers;`
C++: `__property double MinimumIncludingOutliers = {read=GetMinimumIncludingOutliers};`
↳ The minimum value of the data contained in the internal array, including values marked as outliers 95.

Delphi: `property MaximumIncludingOutliers: Double read GetMaximumIncludingOutliers;`
C++: `__property double MinimumIncludingOutliers = {read=GetMinimumIncludingOutliers};`
↳ The maximum value of the data contained in the internal array, including values marked as outliers 95.

Delphi: `property MinimumIdxIncludingOutliers: Integer read GetMinimumIdxIncludingOutliers;`
C++: `__property int MinimumIdxIncludingOutliers = {read=GetMinimumIdxIncludingOutliers, nodefault};`
↳ The index of the minimum value of the data contained in the internal array, including values marked as outliers 95.

Delphi: `property MaximumIdxIncludingOutliers: Integer read GetMaximumIdxIncludingOutliers;`
C++: `__property int MaximumIdxIncludingOutliers = {read=GetMaximumIdxIncludingOutliers, nodefault};`
↳ The index of the maximum value of the data contained in the internal array, including values marked as outliers 95.

Delphi: `property SumIncludingOutliers: Double read GetSumIncludingOutliers;`
C++: `__property SumIncludingOutliers = {read=GetSumIncludingOutliers};`
↳ Returns the sum of all numbers contained in the internal array, including values marked as outliers 95.

Delphi: `property DBData: Boolean read FDBData;`
C++: `__property bool DBData = {read=FDBData, nodefault};`

Returns *true* if a database is connected via DataSource 96 and DataField 96 specifies a valid numerical data field.

Published Properties

Delphi: `property Capacity: Integer read FCapacity write SetCapacity;`
C++: `__property int Capacity = {read=FCapacity, write=SetCapacity, nodefault};`

Size of the array allocated. If you want to store large amounts of data and know the number of points beforehand, you can set the storage capacity accordingly. This will speed up the code and you will have less memory fragmentation.

Delphi: `property Increment: Integer read FIncrement write SetIncrement;`
C++: `__property int Increment = {read=FIncrement, write=SetIncrement, nodefault};`

Size value by which the capacity 96 of the storage array is to be incremented, if the value to be stored does not fit to the actual size of the array.

Delphi: `property PersistentValues: Boolean read FPersistentValues write FPersistentValues;`
C++: `__property bool PersistentValues = {read=FPersistentValues, write=FPersistentValues, nodefault};`

If set to *true*, then the values of the vector generated at design time 22 will be loaded at runtime start.

Delphi: `property DataField: string read GetDataField write SetDataField;`
C++: `__property AnsiString DataField = {read=GetDataField, write=SetDataField};`

Name of the database field used to get the data. Note that only numerical field names are listed in the object inspector. Leave empty if you do not want to link to a database.

Delphi: `property DataSource: TDataSource read GetDataSource write SetDataSource;`
C++: `__property Db::TDataSource* DataSource = {read=GetDataSource, write=SetDataSource};`

Data source to connect to that provides the database supplying the numerical data. Leave empty if you do not want to link to a database.

Delphi: `property InternalArray: TDoubleDynArray read FArray write SetValues;`
C++: `__property TDoubleDynArray InternalArray = {read=FArray, write=SetValues};`

The internal storage array. Please note that reading this property the length of the array is not equal to the count 94 of items. If you assign your own array to this property the count of the items will be set to the length of the array and related series will be update. This might be useful with very large counts of data where performance is an issue.

Public Methods

Delphi: `function Add(const n: Double): Integer;`
C++: `int __fastcall Add(const double n);`

↳ Extends the storage array, giving it the capacity 96 for storing one more point. Sets the last item to the new value ***n***. Returns the new count 94.

Public Methods

Delphi:
```
procedure AddRange(Value: array of Extended); overload;
procedure AddRange(Value: TDoubleDynArray); overload;
procedure AddRange(Value: TSingleDynArray); overload;
procedure AddRange(Value: TIntegerDynArray); overload;
```
C++:
```
void __fastcall AddRange(System::Extended *Value, const int
Value_Size)
void __fastcall AddRange(TDoubleDynArray Value)
void __fastcall AddRange(TSingleDynArray Value)
void __fastcall AddRange(TIntegerDynArray Value)
```

↳ Extends the storage array with all the values passed in the array in one go. Adapts the count 94 accordingly.

Delphi: `function Insert(Idx: Integer; const n: Double): Integer;`
C++: `int __fastcall Insert(int Idx, const double n);`

↳ Inserts a new item ***n*** at index ***Idx***. Moves all items 94 that start with the index one position upwards. Returns new incremented count.

Delphi: `procedure Delete(Idx: Integer);`
C++: `void __fastcall Delete(int Idx);`

↳ Deletes the item at index ***Idx***. Moves all items 94 with a higher index one position downwards. Decrements count 94 by 1.

Delphi: `procedure Clear;`
C++: `void __fastcall Clear(void);`

↳ Clears the vector. Sets count 94 to 0 and clears the outliers 97.

Delphi: `function IsChanged: Boolean; virtual;`
C++: `virtual bool __fastcall IsChanged(void);`

↳ Returns *true* if the vector was altered since the last internal calculations (for internal use only).

Delphi: `procedure AddOutlier(Idx: Integer);`
C++: `void __fastcall AddOutlier(int Idx);`

↳ Mark the value indexed by ***Idx*** as an outlier.

Delphi: `procedure RemoveOutlier(Idx: Integer);`
C++: `void __fastcall RemoveOutlier(int Idx);`

↳ Mark the value indexed by ***Idx*** as a non-outlier.

Delphi: `procedure ClearOutlieres;`
C++: `void __fastcall ClearOutlieres(void);`

↳ Clears the Outliers 95. AnyOutliers 95 will subsequently return *false*.

Delphi: `function IsOutlier(Idx: Integer): Boolean;`
C++: `bool __fastcall IsOutlier(int Idx);`

↳ Returns *true* if the value with index ***Idx*** has been marked as an outlier.

Delphi:
```
procedure ReadFrom(FileName: string); overload;
procedure ReadFrom(Stream: TStream); overload;
procedure ReadFrom(Node: IXMLNode); overload;
procedure ReadFrom(Ini: TCustomIniFile; Section: string);
overload;
```
C++:
```
void __fastcall ReadFrom(AnsiString FileName);
void __fastcall ReadFrom(Classes::TStream* Stream);
void __fastcall ReadFrom(Xmlintf::_di_IXMLNode Node);
void __fastcall ReadFrom(Inifiles::TCustomIniFile* Ini,
AnsiString Section);
```
- The data can be read from a file in four different ways:
  - from a binary file specified by name,
  - from any binary stream,
  - from an XML document node,
  - from a specified section of an INI file.

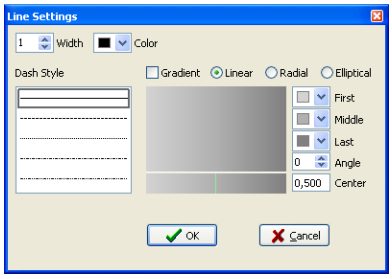
Delphi:
```
procedure WriteTo(FileName: string); overload;
procedure WriteTo(Stream: TStream); overload;
procedure WriteTo(Node: IXMLNode); overload;
procedure WriteTo(Ini: TCustomIniFile; Section: string);
overload;
```
C++:
```
void __fastcall WriteTo(AnsiString FileName);
void __fastcall WriteTo(Classes::TStream* Stream);
void __fastcall WriteTo(Xmlintf::_di_IXMLNode Node);
void __fastcall WriteTo(Inifiles::TCustomIniFile* Ini,
AnsiString Section);
```
- The data can be saved to a file in four different ways:
  - to a binary file specified by name,
  - to any binary stream,
  - to an XML document node,
  - to a specified section of an INI file.

3.5.3 TRtValueTransformation Type

Unit/Namespace: `RtSeries`

Delphi:
```
TRtTransformEventArgs = record
  Value: Double;
  Index: Integer;
end;
TRtValueTransformation = function(Sender: TObject; e:
TRtTransformEventArgs): Double of object;
```
C++:
```
struct TRtTransformEventArgs{
public:
      double Value;
      int Index;
};
typedef double __fastcall (__closure *TRtValueTransformation)
(System::TObject* Sender, const TRtTransformEventArgs &e);
```
- Type used for value transformations in calculations 44 and series. The calling procedure will pass the ***e.Value*** with the non-transformed source value from the vector. The ***e.Index*** is set to the index of the source value. The function must return the transformed result.

3.5.4 TRtSimpleLineSettings Class



This class is used to store the properties of lines to draw - namely, the color, dash style and width - enabling a more structured access. The properties can be set interactively using a special property editor.

Unit/Namespace: `RtStyles`

Declaration

Delphi: `TRtSimpleLineSettings = class(TPersistent);`
C++: `class TRtSimpleLineSettings : public Classes::TPersistent;`

Published Properties

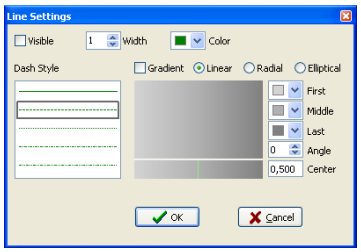
Delphi: `property Color: TRtColor` 58 `read FColor write SetColor;`
C++: `__property Rtgdi::TRtColor Color = {read=FColor, write=SetColor, nodefault};`
↳ The color of the line.

Delphi: `property DashStyle: TRtDashStyle` 58 `read FDashStyle write SetDashStyle;`
C++: `__property Gdipapi::TDashStyle DashStyle = {read=FDashStyle, write=SetDashStyle, default=0};`
↳ The dash style of the line.

Delphi: `property Width: Single read FWidth write SetWidth;`
C++: `__property float Width = {read=FWidth, write=SetWidth};`
↳ The width of the line.

Delphi: `property Gradient: TRtGradientSettings` 64 `read FGradient write GetGradient;`
C++: `__property Rtstyles::TRtGradientSettings* Gradient = {read=FGradient, write=SetGradient};`
↳ Settings for an optional gradient to fill the line instead of the color 99.

3.5.5 TRtLineSettings Class



This class is extends the TRtSimpleLineSettings 99 above with its ***Visible*** property. The properties can be set interactively using a special property editor.

Unit/Namespace: `RtStyles`

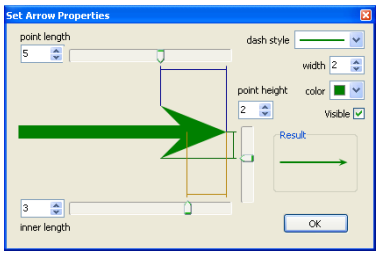
Declaration

Delphi: `TRtLineSettings = class(TRtSimpleLineSettings);`
C++: `class TRtLineSettings : public TRtSimpleLineSettings;`

Published Property

Delphi: `property Visible: Boolean read FVisible write SetVisible;`
C++: `__property bool Visible = {read=FVisible, write=SetVisible, nodefault};`
↳ The visibility of the line.

3.5.6 TRtArrowSettings Class



This class is used to store the additional properties of lines with arrow tips. The properties can be set interactively using a special property editor.

Unit/Namespace: `RtStyles`

Declaration

Delphi: `TRtArrowSettings = class(` TRtLineSettings 99 `);`
C++: `class TRtArrowSettings : public TRtLineSettings;`

Published Properties

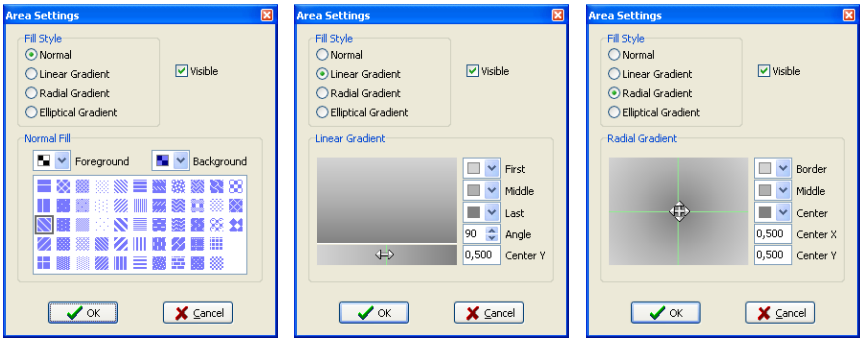
Delphi: `property PointLength: Single read FPointLength write SetPointLength;`
C++: `__property float PointLength = {read=FPointLength, write=SetPointLength};`
↳ The length of the arrow point as a ratio to line width.

Delphi: `property InnerLength: Single read FInnerLength write SetInnerLength;`
C++: `__property float InnerLength = {read=FInnerLength, write=SetInnerLength};`
↳ The length of the inner part of the arrow point as a ratio to line width.

Delphi: `property PointHeight: Single read FPointHeight write SetPointHeight;`
C++: `__property float PointHeight = {read=FPointHeight, write=SetPointHeight};`
↳ The height of the arrow point as a ratio to line width.

3.5.7 TRtAreaSettings Class

This class is used to store the properties of hatch-filled or gradient areas drawn below lines, including their colors and the area style, enabling a more structured access. The properties can be set interactively using a special property editor.



Unit/Namespace: `RtStyles`

Declaration

Delphi: `TRtAreaSettings = class(TPersistent);`
C++: `class TRtAreaSettings : public Classes::TPersistent;`

Published Properties

Delphi: `property Visible: Boolean read FVisible write SetVisible;`
C++: `__property bool Visible = {read=FVisible, write=SetVisible, nodefault};`
↳ The visibility of the filled area.

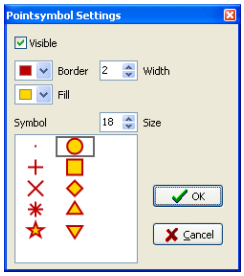
Delphi: `property BackColor: TRtColor` 58 `read FBackColor write SetBackColor;`
C++: `__property Rtgdi::TRtColor BackColor = {read=FBackColor, write=SetBackColor, nodefault};`
↳ The background color of the filled area.

Delphi: `property ForeColor: TRtColor` 58 `read FForeColor write SetForeColor;`
C++: `__property Rtgdi::TRtColor ForeColor = {read=FForeColor, write=SetForeColor, nodefault};`
↳ The foreground color of the filled area.

Delphi: `property AreaStyle: TRtAreaStyle` 59 `read FAreaStyle write SetAreaStyle;`
C++: `__property Rtgdi::TRtAreaStyle AreaStyle = {read=FAreaStyle, write=SetAreaStyle, nodefault};`
↳ The area hatch style of the filled area.

Delphi: `property Gradient: TRtGradientSettings` 64 `read FGradient write FGradient;`
C++: `__property TRtGradientSettings* Gradient = {read=FGradient, write=FGradient};`
↳ The optional gradient settings.

3.5.8 TRtPointSymbolSettings Class



This class is used to store the properties of point symbols to draw, including colors, size, border line width and symbol, enabling a more structured access. The properties can be set interactively using a special property editor.

Unit/Namespace: `RtStyles`

Declaration

Delphi: `TRtPointSymbolSettings = class(TPersistent);`
C++: `class TRtPointSymbolSettings : public Classes::TPersistent;`

Published Properties

Delphi: `property Visible: Boolean read FVisible write SetVisible;`
C++: `__property bool Visible = {read=FVisible, write=SetVisible, nodefault};`
↳ The visibility of the point symbol.

Delphi: `property Size: Single read FSize write SetSize;`
C++: `__property float Size = {read=FSize, write=SetSize};`
↳ The size of the point symbol.

Delphi: `property BorderLineWidth: Single read FBorderLineWidth write SetBorderLineWidth;`
C++: `__property float BorderLineWidth = {read=FBorderLineWidth, write=SetBorderLineWidth};`
↳ The width of the border line of the point symbol.

Delphi: `property BorderColor: TRtColor` 58 `read FBorderColor write SetBorderColor;`
C++: `__property Rtgdi::TRtColor BorderColor = {read=FBorderColor, write=SetBorderColor, nodefault};`
↳ The color of the border line of the point symbol.

Delphi: `property FillColor: TRtColor` 58 `read FFillColor write SetFillColor;`
C++: `__property Rtgdi::TRtColor FillColor = {read=FFillColor, write=SetFillColor, nodefault};`
↳ The fill color of the point symbol.

Delphi: `property PointSymbol: TRtPointSymbol` 63 `read FPointSymbol write SetPointSymbol;`
C++: `__property Rtdrawing::TRtPointSymbol PointSymbol = {read=FPointSymbol, write=SetPointSymbol, nodefault};`
↳ The point symbol type drawn.

3.5.9 TRtSeries Class

This class is the ancestor for all the line-, point-, bar-, arrow-, bubble-, OHLC-, CandleStick- and function series components. It holds the references to the data vectors used by the assigned axes for calculating point positions and provides the methods needed to draw the series in the graph.

Unit/Namespace: `RtSeries`

Declaration

Delphi: `TRtSeries = class(TComponent);`
C++: `class TRtSeries : public Classes::TComponent;`

Public Properties

Delphi: `property XStart: Double read FXStart;`
C++: `__property double XStart = {read=FXStart};`
↳ The minimum of the X-values assigned.

Delphi: `property XStop: Double read FXStop;`
C++: `__property double XStop = {read=FXStop};`
↳ The maximum of the X-values assigned.

Delphi: `property YStart: Double read FYStart;`
C++: `__property double YStart = {read=FYStart};`
↳ The minimum of the Y-values assigned.

Delphi: `property YStop: Double read FYStop;`
C++: `__property double YStop = {read=FYStop};`
↳ The maximum of the Y-values assigned.

Delphi: `property Count: Integer read FCount;`
C++: `__property int Count = {read=FCount, nodefault};`
↳ The number of points to display.

Delphi: `property Points: TRtPointVector` 93 `read FPoints;`
C++: `__property int Count = {read=FCount, nodefault};`
↳ The calculated point positions to display in screen (world) coordinates.

Delphi: `property LinePen: TRtPen` 58 `read FPen write SetPen;`
C++: `__property Rtgdi::TRtPen* LinePen = {read=FPen, write=SetPen};`
↳ The pen object used to draw the lines in the graph.

Delphi: `property AreaBrush: TRtBrush` 58 `read FBrush write SetBrush;`
C++: `__property Gdipobj::TGPBrush* AreaBrush = {read=FBrush, write=SetBrush};`
↳ The brush object used to draw the filled area below the lines in the graph.

Delphi: `property Layer: Integer read FLayer write FLayer;`
C++: `__property int Layer = {read=FLayer, write=FLayer, nodefault};`
↳ The layer used for display order (for internal use only).

Published Properties

Delphi: `property Visible: Boolean read FVisible write SetVisible;`
C++: `__property bool Visible = {read=FVisible, write=SetVisible, nodefault};`

The visibility of the series. Note that, the visibility of the line, area, point symbols, etc., can be controlled separately.

Delphi: `property AutoUpdate: Boolean read FAutoUpdate write FAutoUpdate;`
C++: `__property bool AutoUpdate = {read=FAutoUpdate, write=FAutoUpdate, nodefault};`

If set to *true*, the series will be automatically updated when a value is added, changed or removed via the X- or YData vectors.

Delphi: `property YAxis: TRtAxis` 138 `read FYAxis write SetYAxis;`
C++: `__property Rtaxis::TRtAxis* YAxis = {read=FYAxis, write=SetYAxis};`

The reference to the y-axis, giving the scaling for displaying the points.

Delphi: `property XData: TRtCustomDoubleVector` 94 `read FXData write SetXData;`
C++: `__property Rtvectors::TRtCustomDoubleVector* XData = {read=FXData, write=SetXData};`

The reference to the storage vector of the X-data for calculating the points. If left as as *nil* and ***YData*** has been set, the point index will be taken as X-values. This will be indicated as (Index) in the object inspector.

Delphi: `property YData: TRtCustomDoubleVector` 94 `read FYData write SetYData;`
C++: `__property Rtvectors::TRtCustomDoubleVector* YData = {read=FYData, write=SetYData};`

The reference to the storage vector of the Y-data for calculating the points. If left as *nil* and ***XData*** has been set, the point index will be taken as Y-values. This will be indicated as (Index) in the object inspector.

Delphi: `property Line: TRtLineSettings` 99 `read FLineSettings write FLineSettings;`
C++: `__property Rtstyles::TRtLineSettings* Line = {read=FLineSettings, write=FLineSettings};`

The color, dash style and width settings for the line to be drawn in the graph.

Delphi: `property Area: TRtAreaSettings read FAreaSettings` 101 `write FAreaSettings;`
C++: `__property Rtstyles::TRtAreaSettings* Area = {read=FAreaSettings, write=FAreaSettings};`

The settings for colors and hatch area style to draw the filled area below the lines in the graph.

Delphi: `property Caption: TRtRichCaption` 70 `read FCaption write SetCaption;`
C++: `__property WideString Caption = {read=FCaption, write=SetCaption};`

The caption of the series, to be shown in the legend 35.

Delphi: `property IncludeInLegend: Boolean read FIncludeInLegend write SetIncludeInLegend;`

C++: `__property bool IncludeInLegend = {read=FIncludeInLegend, write=SetIncludeInLegend, nodefault};`

↳ Switches the display of the series item on or off in the legend 35 series list.

Delphi: `property MergeLegendItemWith: TRtSeries 103 read FMergeLegendItemWith write SetMergeLegendItemWith;`

C++: `__property TRtSeries MergeLegendItemWith = {read=FMergeLegendItemWith, write=SetMergeLegendItemWith, nodefault};`

↳ The reference to the parent series, giving the caption and other display items for the legend. You should set IncludeInLegend 105 to *false* when using this option.

Delphi: `property OutliersVisible: Boolean read FOutliersVisible write SetOutliersVisible;`

C++: `__property bool OutliersVisible = {read=FOutliersVisible, write=SetOutliersVisible, nodefault};`

↳ If set to *false*, the series line will not display any X/Y-data points marked as outliers 95. If set *true*, all points available will be used for display.

Events

Delphi: `property XValueTransformation: TRtValueTransformation 98 read mXValueTransformation write SetXValueTransformation;`

C++: `__property TRtValueTransformation XValueTransformation = {read=mXValueTransformation, write=SetXValueTransformation};`

↳ This event is triggered each time an X-value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `property YValueTransformation: TRtValueTransformation 98 read mYValueTransformation write SetYValueTransformation;`

C++: `__property TRtValueTransformation YValueTransformation = {read=mYValueTransformation, write=SetYValueTransformation};`

↳ This event is triggered each time an Y-value is needed for calculation. You can use it to return any other transformed value instead.

Delphi:
```
TRtPenEventArgs = record
  Index: Integer;
  X, Y: Double;
  Pen: TRtPen 58;
  Visible: Boolean;
end;
TRtPenEventHandler = procedure(Sender: TObject; var e: TRtPenEventArgs) of object;
property OnGetPen: TRtPenEventHandler read mGetPen write mGetPen;
```

C++:
```
struct TRtPenEventArgs{
public:
      int Index;
      double X;
      double Y;
      Rtgdi::TRtPen* Pen;
      bool Visible;
};
typedef void __fastcall (__closure *TRtPenEventHandler)
(System::TObject* Sender, TRtPenEventArgs &e);
```

```
__property TRtPenEventHandler OnGetPen = {read=mGetPen,
write=mGetPen};
```

This event can be used to customize the style of single line segments in the series line. If specified, it will be called when drawing the line for each data point.

Delphi:
```
TRtBrushEventArgs = record
  Index: Integer;
  X,Y: Double;
  Brush: TRtBrush 58;
  Visible: Boolean;
end;
TRtBrushEventHandler = procedure(Sender: TObject; var e:
TRtBrushEventArgs) of object;
property OnGetBrush: TRtBrushEventHandler read mGetBrush
write mGetBrush;
```

C++:
```
struct TRtBrushEventArgs{
public:
      int Index;
      double X;
      double Y;
      Gdipobj::TGPBrush* Brush;
      bool Visible;
};
typedef void __fastcall (__closure *TRtBrushEventHandler)
(System::TObject* Sender, TRtBrushEventArgs &e);
__property TRtBrushEventHandler OnGetBrush = {read=mGetBrush,
write=mGetBrush};
```

This event can be used to customize the style of single area below the line segments in the series line. If specified, it will be called when filling the area on each data point.

Public Methods

Delphi:
```
function GetY(AtX: System.Double): System.Double; virtual;
```

C++:
```
virtual double __fastcall GetY(double AtX);
```

This function returns the Y-value for any X-position. In normal line series this will be linearly interpolated between the next data points; with function series this will give the calculated function result at the X-position.

Delphi:
```
TRtNearestPointResult = record
  Found: Boolean;
  Distance: Single;
  Index: Integer;
  X,Y: Double;
  ThePoint: TRtPoint 59;
end;
function NearestPoint(FromPosition: TPoint):
TRtNearestPointResult;
```

C++:
```
struct TRtNearestPointResult{
public:
      bool Found;
      float Distance;
      int Index;
      double X;
      double Y;
      Gdipapi::TGPPointF ThePoint;
};
```

```
TRtNearestPointResult __fastcall NearestPoint(const Types::
TPoint &FromPosition);
```

This function can be used to search the series for the next data point relative to a mouse position supplied with the ***FromPosition*** parameter. The search is only performed within the current display range. When zoomed, only the non-clipped data can be found. The result will contain information if any point was found, namely: the distance in screen pixels to the mouse position, the index of the data point found, the X and Y data values and the screen coordinates of the point. This method is used internally for the snapping option of markers 37.

Delphi: `function NearestX(FromX: Integer): TRtNearestPointResult`;
C++: `TRtNearestPointResult __fastcall NearestX(int FromX);`

This function can be used to search the series for the next data point relative to a mouse X-position supplied with the ***FromPosition*** parameter. The search is only performed within the current display range. When zoomed, only the non-clipped data can be found. The result will contain information if any point was found, namely: the distance in screen pixels to the mouse position, the index of the data point found, the X and Y data values and the screen coordinates of the point. This method is used internally for the snapping option of vertical markers 37.

Delphi: `function NearestY(FromY: Integer): TRtNearestPointResult`;
C++: `TRtNearestPointResult __fastcall NearestY(int FromY);`

This function can be used to search the series for the next data point relative to a mouse Y-position supplied with the ***FromPosition*** parameter. The search is only performed within the current display range. When zoomed, only the non-clipped data can be found. The result will contain information if any point was found, namely: the distance in screen pixels to the mouse position, the index of the data point found, the X and Y data values and the screen coordinates of the point. This method is used internally for the snapping option of horizontal markers 37.

Delphi:
```
function FindPoint(X,Y: Integer; Delta: Single): Integer;
overload; virtual;
function FindPoint(X,Y: Integer): Integer; overload; virtual;
```
C++:
```
virtual int __fastcall FindPoint(int X, int Y, float Delta);
virtual int __fastcall FindPoint(int X, int Y);
```

This function can be used to search the series for the data point under the mouse position supplied as an X,Y parameter. If the ***Delta*** parameter was supplied, the function will search points around this distance. If you omit this parameter, the function will search the actual size of the points display area. This can be especially interesting to use for updating a status display of values under the cursor. Used internally with the TRtGraph2D.FindSeriesPoint 160 method, which searches the whole graph and also detects the relevant series.

Delphi: `procedure Invalidate;`
C++: `void __fastcall Invalidate(void);`

Marks the series as needing to be redrawn (for internal use only).

Delphi: `procedure RangesModified;`
C++: `void __fastcall RangesModified(void);`

Marks the assigned axes ranges as needing to be updated, and the series points as needing to be recalculated and redrawn (for internal use only).

Delphi: `procedure PointsModified;`
C++: `void __fastcall PointsModified(void);`

Marks the series points as needing to be recalculated and redrawn (for internal use only).

Delphi: `procedure Calculate; virtual;`
C++: `virtual void __fastcall Calculate(void);`

Calls the internal calculations of the series. Checks for changes with the assigned data and takes care of any recalculations needed.Use directly only when the AutoUpdate 104 has been set to *false*.

Delphi:
```
procedure WriteTo(FileName: string); overload;
procedure WriteTo(Stream: TStream); overload; virtual;
procedure WriteTo(Node: IXMLNode); overload;
procedure WriteTo(Ini: TCustomIniFile; Section: string);
overload; virtual;
```
C++:
```
void __fastcall WriteTo(AnsiString FileName);
virtual void __fastcall WriteTo(Classes::TStream* Stream);
void __fastcall WriteTo(Xmlintf::_di_IXMLNode Node);
virtual void __fastcall WriteTo(Inifiles::TCustomIniFile*
Ini, AnsiString Section);
```

The series settings can be saved to a file in four different ways:
to a binary file specified by name,
to any binary stream,
to an XML document node,
to a specified section of an INI-file.

Delphi:
```
procedure ReadFrom(FileName: string); overload;
procedure ReadFrom(Stream: TStream); overload; virtual;
procedure ReadFrom(Node: IXMLNode); overload; virtual;
procedure ReadFrom(Ini: TCustomIniFile; Section: string);
overload; virtual;
```
C++:
```
void __fastcall ReadFrom(AnsiString FileName);
virtual void __fastcall ReadFrom(Classes::TStream* Stream);
virtual void __fastcall ReadFrom(Xmlintf::_di_IXMLNode Node);
virtual void __fastcall ReadFrom(Inifiles::TCustomIniFile*
Ini, AnsiString Section);
```

The series settings can be read from a file in four different ways:
from a binary file specified by name,
from any binary stream,
from an XML document node,
from a specified section of an INI file.

3.5.10 TRtLineSeries Class

This class is derived from TRtSeries 103 and publishes all the properties needed to display a simple poly line with the graph.

Unit/Namespace: `RtSeries`

Declaration

Delphi: `TRtLineSeries = class(TRtSeries);`
C++: `class TRtLineSeries : public TRtSeries;`

Published Property

Delphi: `property DrawBandTo: TRtLineSeries 108 read FDrawBandTo write SetDrawBandTo;`

C++: `__property TRtLineSeries DrawBandTo = {read=FDrawBandTo, write=SetDrawBandTo, nodefault};`

The reference to another series to draw a filled area as a band.

3.5.11 TRtPointSeries Class

This class is derived from TRtLineSeries 108 and adds properties and methods to draw point symbols with the series line.

Unit/Namespace: `RtSeries`

Declaration

Delphi: `TRtPointSeries = class(TRtLineSeries)`

C++: `class TRtPointSeries : public TRtLineSeries;`

Public Properties

Delphi: `property SymbolBorderPen: TRtPen 58 read FSymbolPen write SetSymbolPen;`

C++: `__property Rtgdi::TRtPen* SymbolBorderPen = {read=FSymbolPen, write=SetSymbolPen}`

The pen object used to draw the border line of the point symbol in the graph.

Delphi: `property SymbolFillBrush: TRtBrush 58 read FSymbolBrush write SetSymbolBrush;`

C++: `__property Gdipobj::TGPBrush* SymbolFillBrush = {read=FSymbolBrush, write=SetSymbolBrush};`

The brush object used to draw the filled area inside the point symbol in the graph.

Published Property

Delphi: `property PointSymbol: TRtPointSymbolSettings 102 read FPointSymbolSettings write FPointSymbolSettings;`

C++: `__property Rtstyles::TRtPointSymbolSettings* PointSymbol = {read=FPointSymbolSettings, write=FPointSymbolSettings};`

The settings of the point symbol.

Event

Delphi:
```
PointSymbolEventArgs = record
  Index: Integer;
  X,Y: Double;
  Size: Single;
  Pen: TRtPen 58;
  Brush: TRtBrush 58;
  Symbol: TRtPointSymbol 63;
  Visible: Boolean;
end;
PointSymbolEventHandler = procedure(Sender: TObject; var e:
PointSymbolEventArgs) of object;
property OnGetPointSymbol: PointSymbolEventHandler read
mPointSymbol write mPointSymbol;
```

C++:
```
struct PointSymbolEventArgs{
public:
      int Index;
      double X;
      double Y;
      float Size;
      Rtgdi::TRtPen* Pen;
      Gdipobj::TGPBrush* Brush;
      Rtdrawing::TRtPointSymbol Symbol;
      bool Visible;
};
typedef void __fastcall (__closure *PointSymbolEventHandler)
(System::TObject* Sender, PointSymbolEventArgs &e);
__property PointSymbolEventHandler OnGetPointSymbol =
{read=mPointSymbol, write=mPointSymbol};
```

This event can be used to customize the style of single data points in the series. If specified, it will be called when drawing the symbol on each data point.

3.5.12 TRtPointWithErrorSeries Class

This class is derived from TRtPointSeries 109 and adds properties and methods to draw error indicators with the point symbols.

Unit/Namespace: `RtSeries`

Declaration

Delphi: `TRtPointWithErrorSeries = class(TRtPointSeries)`

C++: `class TRtPointWithErrorSeries : public TRtPointSeries;`

Published Properties

Delphi: `property ErrorIndicator: TRtLineSettings 99 read FErrorIndicator write FErrorIndicator;`

C++: `__property Rtstyles::TRtLineSettings* ErrorIndicator = {read=FErrorIndicator, write=FErrorIndicator};`

The settings for color, dash style and width of the error indicators line to be drawn at the point symbols.

Delphi: `property DeltaXPlus: TRtCustomDoubleVector 94 read FdXPlus write SetDeltaXPlus;`

C++: `__property Rtvectors::TRtCustomDoubleVector* DeltaXPlus = {read=FdXPlus, write=SetDeltaXPlus};`

The reference to the storage vector that holds the length of the error indicator pointing right of the X-data point. If no vector is assigned, the indicator is not drawn.

Delphi: `property DeltaXMinus: TRtCustomDoubleVector 94 read FdXMinus write SetDeltaXMinus;`

C++: `__property Rtvectors::TRtCustomDoubleVector* DeltaXMinus = {read=FdXMinus, write=SetDeltaXMinus};`

The reference to the storage vector that holds the length of the error indicator pointing left of the X-data point. If no vector is assigned, the indicator is not drawn.

Delphi: `property DeltaYPlus: TRtCustomDoubleVector` 94 `read FdYPlus write SetDeltaYPlus;`
C++: `__property Rtvectors::TRtCustomDoubleVector* DeltaYPlus = {read=FdYPlus, write=SetDeltaYPlus};`

The reference to the storage vector that holds the length of the error indicator pointing upwards of the Y-data point. If no vector is assigned, the indicator is not drawn.

Delphi: `property DeltaYMinus: TRtCustomDoubleVector` 94 `read FdYMinus write SetDeltaYMinus;`
C++: `__property Rtvectors::TRtCustomDoubleVector* DeltaYMinus = {read=FdYMinus, write=SetDeltaYMinus};`

The reference to the storage vector that holds the length of the error indicator pointing downwards of the Y-data point. If no vector is assigned, the indicator is not drawn.

Events

Delphi: `DxPlusTransformation: TRtValueTransformation` 98 `read mGetDxPlusTransformation write SetDxPlusTransformation;`
C++: `__property TRtValueTransformation DxPlusTransformation = {read=mGetDxPlusTransformation, write=SetDxPlusTransformation};`

This event is triggered each time a DeltaXPlus 110-value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `DxMinusTransformation: TRtValueTransformation` 98 `read mGetDxMinusTransformation write SetDxMinusTransformation;`
C++: `__property TRtValueTransformation DxMinusTransformation = {read=mGetDxMinusTransformation, write=SetDxMinusTransformation};`

This event is triggered each time a DeltaXMinus 110-value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `DyPlusTransformation: TRtValueTransformation` 98 `read mGetDyPlusTransformation write SetDxPlusTransformation;`
C++: `__property TRtValueTransformation DyPlusTransformation = {read=mGetDyPlusTransformation, write=SetDyPlusTransformation};`

This event is triggered each time a DeltaYPlus 111-value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `DyMinusTransformation: TRtValueTransformation` 98 `read mGetDyMinusTransformation write SetDxPlusTransformation;`
C++: `__property TRtValueTransformation DyMinusTransformation = {read=mGetDyMinusTransformation, write=SetDyMinusTransformation};`

This event is triggered each time a DeltaYMinus 111-value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `property OnGetIndicatorsPen: TRtPenEventHandler` 105 `read mGetPen write mGetPen;`
C++: `__property TRtPenEventHandler OnGetIndicatorsPen = {read=mGetPen, write=mGetPen};`

This event can be used to customize the style of the single point symbols error indicator line. If specified, it will be called when drawing the indicators on each data point.

3.5.13 TRtCaptions Class

Delphi:
```
TRtCaptions = TStrings;          // VCL.net and Studio 2009
TRtCaptionList = TStringList;
TRtCaptions = TTntStrings;       // Win32
TRtCaptionList = TTntStringList;
```
C++:
```
typedef TTntStrings TRtCaptions;
typedef TTntStringList TRtCaptionList;
```
This class is used to store optional captions used for the rubrics 145 of the axis or bars labels 113 in a wide string list.

3.5.14 TRtBars Class

This class is derived from TRtLineSeries 108 and provides properties and methods needed to display bar charts with the graph.

Unit/Namespace: `RtBars`

Declaration

Delphi:
```
TRtBars = class(TRtSeries);
```
C++:
```
class TRtBars : public Rtseries::TRtSeries;
```

Published Properties

Delphi:
```
TRtBarStyle=(Flat,Cuboid,Cylinder);
property BarStyle: TRtBarStyle read FBarStyle write
SetBarsStyle;
```
C++:
```
enum TRtBarStyle { Flat, Cuboid, Cylinder };
__property TRtBarStyle BarStyle = {read=FBarStyle,
write=SetBarsStyle, nodefault};
```
The bar can be drawn as flat rectangle, cuboid or cylinder. This property sets this drawing style.

Delphi:
```
property BarsHorizontal: Boolean read FBarsHorizontal write
SetBarsHorizontal;
```
C++:
```
__property bool BarsHorizontal = {read=FBarsHorizontal,
write=SetBarsHorizontal, nodefault};
```
The bars can be drawn vertically or horizontally. If set to *true*, the bars are drawn horizontally. Changing this property will change all other bar series with the related graph to the same setting. XData 103 and YData 104 vectors will also be swapped if the orientation changes.

Delphi:
```
TRtBarsOverlayStyle=(Shift,Stack);
property OverlayStyle: TRtBarsOverlayStyle read FOverlayStyle
write SetOverlayStyle;
```
C++:
```
enum TRtBarsOverlayStyle { Shift, Stack };
__property TRtBarsOverlayStyle OverlayStyle =
{read=FOverlayStyle, write=SetOverlayStyle, nodefault};
```
If more then one bar series is displayed, the series can be overlaid in two different ways. If set to *Shift*, the series will share the width of the classes and will be drawn side-by-side, shifted in relation to each other. If set to *Stack*, they will be drawn stacked on top each other, using the whole class width.

Delphi: `property Depth3D: Double read F3DDepth write Set3DDepth;`
C++: `__property double Depth3D = {read=F3DDepth, write=Set3DDepth};`

The depth to draw the 3D effect for cuboids and cylinders, as a percent of the class width.

Delphi: `property ClassesDistance: Double read FClassesDistance write SetClassesDistance;`
C++: `__property double ClassesDistance = {read=FClassesDistance, write=SetClassesDistance};`

The distance of the bars to the class border, as a percent of the class width.

Delphi:
```
TRtBarLabelPosition=(Invisible, OnBottom, Centered, GoldenCut, ThreeFourth, InTop, OnTopOfBar);
property LabelPosition: TRtBarLabelPosition read FLabelPosition write SetLabelPosition;
```
C++:
```
enum TRtBarLabelPosition { Invisible, OnBottom, Centered, GoldenCut, ThreeFourth, InTop, OnTopOfBar };
__property TRtBarLabelPosition LabelPosition = {read=FLabelPosition, write=SetLabelPosition, nodefault};
```

Bars can contain a label indicating their value, or any other caption. The label will display only if the position has been set higher than *Invisible*. If the position is set lower than *OnTopOfBar* and the size of the label does not fit to the bar, it will be drawn on top of the bar.

Delphi:
```
TRtLabelStyle=(Value, CaptionsList, Percent, CaptionAndValue, CaptionAndPercent);
property LabelFrom: TRtLabelStyle read FLabelFrom write SetLabelFrom;
```
C++:
```
enum TRtLabelStyle { Value, CaptionsList, Percent, CaptionAndValue, CaptionAndPercent };
__property TRtLabelStyle LabelFrom = {read=FLabelFrom, write=SetLabelFrom, nodefault};
```

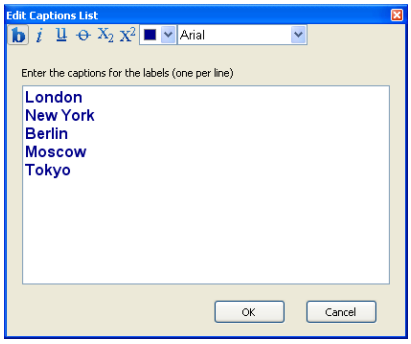
If *Value* has been set, the label displayed will show the numerical value of the bar. The display can be altered using the LabelFormat 113 property. If *CaptionsList* is set, the label retrieves the text from the Captions 113 string list. If *Percent* is selected the text will display the value as percent of the sum of values using the optional LabelFormat. CaptionAndValue and CaptionAndPercent will display combination of both.

Delphi: `property LabelFormat: string read FLabelFormat write SetLabelFormat;`
C++: `__property AnsiString LabelFormat = {read=FLabelFormat, write=SetLabelFormat};`

If LabelFrom 113 has been set to *Value* or *Percent*, the label displayed will show the numerical value of the bar. The format is used in the same way as with the ***FormatFloat*** function. A value "1.234" with a format of "0.00" will therefore display "1.23". An empty format string will display all significant digits.

Delphi: `property Captions: TRtCaptions 112 read GetCaptionsList write SetCaptionsList;`
C++: `__property Tntclasses::TTntStrings* Captions = {read=GetCaptionsList, write=SetCaptionsList};`

If LabelFrom 113 has been set to *CaptionsList*, the labels displayed at the bars retrieve their text from a string list.



A property editor is provided for editing all the captions, available when clicking the ellipsis button in the object inspector with enhanced styles 10.

Delphi: `property Font: TFont read FFont write SetFont;`
C++: `__property Graphics::TFont* Font = {read=FFont, write=SetFont};`
↳ The font used for the optional labels.

Delphi: `property LabelVertical: Boolean read FLabelVertical write SetLabelVertical;`
C++: `__property bool LabelVertical = {read=FLabelVertical, write=SetLabelVertical, nodefault};`
↳ If set to *true*, the optional labels will be drawn vertically.

Delphi: `property float LabelSize = {read=FLabelSize, write=SetLabelSize};`
C++: `__property float LabelSize = {read=FLabelSize, write=SetLabelSize};`
↳ The font size of the optional labels as a percent of the class width. The class does not use absolute sizes: this ensures that the bars look the same even when the graph has been zoomed or more data is subsequently added and the class width shrinks.

Delphi: `property LabelColor: TRtColor read FLabelColor write SetLabelColor;`
C++: `__property Rtgdi::TRtColor LabelColor = {read=FLabelColor, write=SetLabelColor, nodefault};`
↳ The color of the optional labels.

Delphi: `property DataSource: TDataSource read GetDataSource write SetDataSource;`
C++: `__property Db::TDataSource* DataSource = {read=GetDataSource, write=SetDataSource};`
↳ Data source to connect to, providing the database supplying the captions for the optional labels. Leave empty if you do not want to link to a database.

Delphi: `property CaptionsField: string read GetDataField write SetDataField;`
C++: `__property AnsiString CaptionsField = {read=GetDataField, write=SetDataField};`
↳ Name of the database field used to get the captions for the optional labels. Leave empty if you do not want to link to a database.

Event

Delphi:
```
TRtGetLabelEventArgs=record
  Caption: TRtRichCaption;
  Index: Integer;
end;
```

```
TRtGetLabelEventHandler = procedure( Sender: TObject; var e:
TRtGetLabelEventArgs) of object;
property OnGetLabel: TRtGetLabelEventHandler read mGetLabel
write mGetLabel;
```

C++:
```
struct TRtGetLabelEventArgs{
public:
      WideString Caption;
      int Index;
};
typedef void __fastcall ( __closure *TRtBarLabelEventHandler)
( System::TObject* Sender, TRtGetLabelEventArgs &e);
__property TRtGetLabelEventHandler OnGetLabel =
{ read=mGetLabel, write=mGetLabel};
```

This event is triggered each time an optional label caption is required. You can use this to implement your own formatting, for example. The parameter passed gives access to the default caption and the index of the current bar. You can alter the ***e. Caption*** to your needs.

3.5.15 TRtBubbles Class

This class derived from TRtLineSeries 108 provides properties and methods needed to display bubble charts with the graph.

Unit/Namespace: `RtBubbles`

Declaration

Delphi:
```
TRtBubbles = class( TRtSeries);
```

C++:
```
class TRtBubbles : public Rtseries::TRtLineSeries;
```

Published Properties

Delphi:
```
property RadiusData: TRtCustomDoubleVector read FRadius
write SetRadius;
```
94

C++:
```
__property Rtvectors::TRtCustomDoubleVector* RadiusData =
{ read=FRadius, write=SetRadius};
```

The reference to the storage vector of the radius data for calculating the bubble circle or ellipse.

Delphi:
```
property Elliptical: Boolean read FElliptical write
SetElliptical;
```

C++:
```
__property bool Elliptical = { read=FElliptical,
write=SetElliptical, nodefault};
```

If set to *true*, the bubbles will be drawn as an ellipse. The semimajor axis will use the RadiusData 115 and the X-axis scaling. The semiminor axis will use the RadiusData 115 using the Y-axis scaling. If set to *false,* the bubbles will be drawn as circles using the RadiusData 115 with the Y-axis scaling.

Event

Delphi: `property RadiusTransformation: TRtValueTransformation read mRadiusTransformation write SetRadiusTransformation;`

C++: `__property Rtseries::TRtValueTransformation RadiusTransformation = {read=mRadiusTransformation, write=SetRadiusTransformation};`

This event is triggered each time a radius value is needed for calculation. You can use it to return any other transformed value instead.

3.5.16 TRtArrows Class

This class derived from TRtSeries provides properties and methods needed to display 2D vector fields as arrows with the graph.

Unit/Namespace: `RtArrows`

Declaration

Delphi: `TRtArrows = class(TRtSeries);`

C++: `class TRtArrows : public Rtseries::TRtSeries;`

Published Properties

Delphi: `property DeltaX: TRtCustomDoubleVector read FdX write SetDeltaX;`

C++: `__property Rtvectors::TRtCustomDoubleVector* DeltaY = {read=FdY, write=SetDeltaX};`

The reference to a data vector holding the values for the X-component of the arrow direction and length.

Delphi: `property DeltaY: TRtCustomDoubleVector read FdY write SetDeltaY;`

C++: `__property Rtvectors::TRtCustomDoubleVector* DeltaY = {read=FdY, write=SetDeltaY};`

The reference to a data vector holding the values for the Y-component of the arrow direction and length.

Delphi: `property Arrow: TRtArrowSettings read FArrow write FArrow;`

C++: `__property Rtstyles::TRtArrowSettings* Arrow = {read=FArrow, write=FArrow};`

The style of the arrows to draw: color, dash style, line width and point dimensions.

Events

Delphi: `property XDeltaTransformation: TRtValueTransformation read mXDeltaTransformation write SetXDeltaTransformation;`

C++: `__property Rtseries::TRtValueTransformation XDeltaTransformation = {read=mXDeltaTransformation, write=SetXDeltaTransformation};`

This event is triggered each time a Delta-X value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `property YDeltaTransformation: TRtValueTransformation read mYDeltaTransformation write SetYDeltaTransformation;`
C++: `__property Rtseries::TRtValueTransformation YDeltaTransformation = {read=mYDeltaTransformation, write=SetYDeltaTransformation};`
↳ This event is triggered each time a Delta-Y value is needed for calculation. You can use it to return any other transformed value instead.

3.5.17 TRtCustomOHLC Class

This class derived from TRtSeries is the ancestor of the financial chart series TRtOHLC and TRtCandleSticks. It provides all the common properties and methods for drawing symbols expressing opening, high, low and closing prices.

Unit/Namespace: `RtOHLC`

Declaration

Delphi: `TRtCustomOHLC = class(TRtSeries);`
C++: `class TRtCustomOHLC : public Rtseries::TRtSeries;`

Published Properties

Delphi: `property OpenData: TRtCustomDoubleVector read FOpenData write SetOpenData;`
C++: `__property Rtvectors::TRtCustomDoubleVector* OpenData = {read=FOpenData, write=SetOpenData};`
↳ The reference to the storage vector of the opening prices data for calculating the bars.

Delphi: `property HighData: TRtCustomDoubleVector read FHighData write SetHighData;`
C++: `__property Rtvectors::TRtCustomDoubleVector* HighData = {read=FHighData, write=SetHighData};`
↳ The reference to the storage vector of the highest share price data for calculating the bars.

Delphi: `property LowData: TRtCustomDoubleVector read FLowData write SetLowData;`
C++: `__property Rtvectors::TRtCustomDoubleVector* LowData = {read=FLowData, write=SetLowData};`
↳ The reference to the storage vector of the lowest share price data for calculating the bars.

Delphi: `property CloseData: TRtCustomDoubleVector read GetCloseData write SetCloseData;`
C++: `__property Rtvectors::TRtCustomDoubleVector* CloseData = {read=GetCloseData, write=SetCloseData};`
↳ The reference to the storage vector of the closing share price data for calculating the bars.

Delphi: `property BarWidth: Single read FBarWidth write SetBarWidth;`
C++: `__property float BarWidth = {read=FBarWidth, write=SetBarWidth};`
↳ The width of the bars.

Events

Delphi: `property OpenValueTransformation: TRtValueTransformation` 98
`read mOpenValueTransformation write SetOpenValueTransformation;`

C++: `__property Rtseries::TRtValueTransformation OpenValueTransformation = {read=mOpenValueTransformation, write=SetOpenValueTransformation};`

This event is triggered each time an opening price data value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `property HighValueTransformation: TRtValueTransformation` 98
`read mHighValueTransformation write SetHigh ValueTransformation;`

C++: `__property Rtseries::TRtValueTransformation HighValueTransformation = {read=mHighValueTransformation, write=SetHighValueTransformation};`

This event is triggered each time a highest price data value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `property LowValueTransformation: TRtValueTransformation` 98
`read mLowValueTransformation write SetLowValueTransformation;`

C++: `__property Rtseries::TRtValueTransformation LowValueTransformation = {read=mLowValueTransformation, write=SetLowValueTransformation};`

This event is triggered each time a lowest price data value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `property CloseValueTransformation: TRtValueTransformation` 98
`read GetCloseValueTransformation write SetClose ValueTransformation;`

C++: `__property Rtseries::TRtValueTransformation CloseValueTransformation = {read=GetCloseValueTransformation, write=SetCloseValueTransformation};`

This event is triggered each time a closing price data value is needed for calculation. You can use it to return any other transformed value instead.

3.5.18 TRtOHLC Class

This class derived from TRtCustomOHLC 117 provides properties and methods needed to display OHLC (Western plot) charts within the graph.

Unit/Namespace: `RtOHLC`

Declaration

Delphi: `TRtOHLC = class(TRtCustomOHLC);`

C++: `class TRtOHLC : public TRtCustomOHLC;`

Published Property

Delphi: `property IndicatorLength: Single read FIndicatorLength write SetIndicatorLength;`

C++: `__property float IndicatorLength = {read=FIndicatorLength, write=SetIndicatorLength};`

The length of the opening and closing indicators at the sides of the bars.

3.5.19 TRtCandleSticks Class

This class derived from TRtCustomOHLC 117 provides properties and methods needed to display candle sticks charts within the graph.

Unit/Namespace: `RtOHLC`

Declaration

Delphi: `TRtCandleSticks = class(TRtCustomOHLC);`
C++: `class TRtCandleSticks : public TRtCustomOHLC;`

Published Properties

Delphi: `property BullColor: TRtColor 58 read FBullColor write SetBullColor;`
C++: `__property Rtgdi::TRtColor BullColor = {read=FBullColor, write=SetBullColor, nodefault};`
The fill color of the bars, if the share price is rising.

Delphi: `property BearColor: TRtColor 58 read FBearColor write Set BearColor;`
C++: `__property Rtgdi::TRtColor BearColor = {read=FBearColor, write=SetBearColor, nodefault};`
The fill color of the bars, if the share price is falling.

Delphi: `property ShadowsSameColorAsFill: Boolean read FShadowsSameColorAsFill write SetShadowsSameColorAsFill;`
C++: `__property bool ShadowsSameColorAsFill = {read=FShadowsSameColorAsFill, write=SetShadowsSameColorAsFill, nodefault};`
If set to *true*, the color of the shadows indicating highest and lowest share price data will be drawn in the same color as the bar fill. Otherwise they will be drawn using the normal line 104 settings.

3.5.20 TRtFunctionSeries Class

This class is the ancestor of all calculated line components. It defines properties and methods common to all calculated function lines.

Unit/Namespace: `RtFunctionSeries`

Declaration

Delphi: `TRtFunctionSeries = class( TRtLineSeries 108 );`
C++: `class TRtFunctionSeries : public Rtseries::TRtLineSeries;`

Public Property

Delphi: `property Formula: TRtRichCaption 70 read GetFormula;`
C++: `__property WideString Formula = {read=GetFormula};`
The formula used for calculation. Use a TRtRichLabel 71 component to display.

Published Properties

Delphi: `property Weight: TRtCustomDoubleVector read FWeight write SetWeight;`
C++: `__property Rtvectors::TRtCustomDoubleVector* Weight = {read=FWeight, write=SetWeight};`

The reference to the storage vector for the weight data - the statistical weights used in the calculation algorithm giving the result function line. The weight data points are understood to mean: "Take this data point Weight[i] times for the calculation result". If no ***Weight*** vector is assigned, all data point will be considered equal ($w[i]=1$).

Delphi: `property AutoStart: Boolean read FAutoStart write SetAutoStart;`
C++: `__property bool AutoStart = {read=FAutoStart, write=SetAutoStart, nodefault};`

Establishes the start of the calculation as being automatically set to the lowest X-value of the source data.

Delphi: `property AutoStop: Boolean read FAutoStop write SetAutoStop;`
C++: `__property bool AutoStop = {read=FAutoStop, write=SetAutoStop, nodefault};`

Establishes the stop of the calculation as being automatically set to the highest X-value.

Delphi: `property CalculationStart: Double read GetCalcStart write SetCalcStart;`
C++: `__property double CalculationStart = {read=GetCalcStart, write=SetCalcStart};`

Sets the X-start of the calculation, if AutoStart is set to *false*.

Delphi: `property CalculationStop: Double read GetCalcStop write SetCalcStop;`
C++: `__property double CalculationStop = {read=GetCalcStop, write=SetCalcStop};`

Sets the X-stop of the calculation, if AutoStop is set to *false*.

Delphi: `property CoupleCalculationAndDisplay: Boolean read FCoupleCalculationAndDisplay write SetCoupleCalculationAndDisplay;`
C++: `__property bool CoupleCalculationAndDisplay = {read=FCoupleCalculationAndDisplay, write=SetCoupleCalculationAndDisplay, nodefault};`

If set to *true*, the ranges for calculation and display of the line are synchronized. If set to *false*, the display range of the line can be set differently using the DisplayStart and DisplayStop properties. This can be used to show extrapolated lines.

Delphi: `property DisplayAutoStart: Boolean read FDisplayAutoStart write SetDisplayAutoStart;`
C++: `__property bool DisplayAutoStart = {read=FDisplayAutoStart, write=SetDisplayAutoStart, nodefault};`

If set to *true* and CoupleCalculationAndDisplay is *false*, the start of the display range of the function line will be automatically set to the lowest X-value of the source data.

Delphi: `property DisplayAutoStop: Boolean read FDisplayAutoStop write SetDisplayAutoStop;`
C++: `__property bool DisplayAutoStop = {read=FDisplayAutoStop, write=SetDisplayAutoStop, nodefault};`

- If set to *true* and CoupleCalculationAndDisplay 120 is *false*, the stop of the display range of the function line will be automatically set to the highest X-value of the source data.

Delphi: `property DisplayStart: Double read GetDisplayStart write SetDisplayStart;`
C++: `__property double DisplayStart = {read=GetDisplayStart, write=SetDisplayStart};`

- If CoupleCalculationAndDisplay 120 and DisplayAutostart 120 are both *false,* the start of the displayed function line can be set using this property. This can be used to show extrapolated lines.

Delphi: `property DisplayStop: Double read GetDisplayStop write SetDisplayStop;`
C++: `__property double DisplayStop = {read=GetDisplayStop, write=SetDisplayStop};`

- If CoupleCalculationAndDisplay 120 and DisplayAutoStop 121 are both *false,* the stop of the displayed function line can be set using this property. This can be used to show extrapolated lines.

Delphi: `property ClipYValues: Boolean read FClipYValues write SetClipYValues;`
C++: `__property bool ClipYValues = {read=FClipYValues, write=SetClipYValues, nodefault};`

- Sometimes the calculated function lines will exceed the range of the source data points. The display of the function line together with the source data point series might grant the function line too much space (with swingers from spline interpolation, for example). If you set this property to *false*, the function lines Y-values of the function lines will not exceed the Y-range of the source points.

Events

Delphi: `property WeightValueTransformation: TRtValueTransformation` 98 `read GetWeightValueTransformation write SetWeightValueTransformation;`
C++: `__property Rtseries::TRtValueTransformation WeightValueTransformation = {read=GetWeightValueTransformation, write=SetWeightValueTransformation};`

- This event is triggered each time a weight data value is needed for calculation. You can use it to return any other transformed value instead. Note: this event is also triggered if Weight 120 is set to *nil*.

Delphi: `property OnCalculated: TNotifyEvent read mCalculated write mCalculated;`
C++: `__property Classes::TNotifyEvent OnCalculated = {read=mCalculated, write=mCalculated};`

- This event is triggered after the internal calculation completes. You can use it to update any status display that shows calculation results.

Delphi: `TRtCalculationErrorHandler = procedure (Sender: TObject; e: Exception) of object;`
`property OnCalculationError: TRtCalculationErrorHandler read mCalculationError write mCalculationError;`

C++: `typedef void __fastcall (__closure *TRtCalculationErrorHandler)(System::TObject* Sender, Sysutils::Exception* e);`
`__property TRtCalculationErrorHandler OnCalculationError = {read=mCalculationError, write=mCalculationError};`

This event is triggered if the internal calculation raises an error exception. You can use this to keep these exceptions silent or to display the exception message in a status bar. If this event is not handled by you, then the standard error dialog popup will be displayed on calculation errors.

3.5.21 TRtLinearRegression Class

This class derived from TRtFunctionSeries 119 is used to display the calculated straight line linear regression 45 $Y = a + bX$.

Unit/Namespace: `RtLinearRegression`

Declaration

Delphi: `TRtLinearRegression = class(TRtFunctionSeries);`

C++: `class TRtLinearRegression : public Rtfunctionseries::TRtFunctionSeries;`

Published Property

Delphi: `property FixedOffset: Boolean read GetFixedOffset write SetFixedOffset;`

C++: `__property bool FixedOffset = {read=GetFixedOffset, write=SetFixedOffset, nodefault};`

If set to *false*, the regression line ($Y = a + bX$) will be calculated by the method of least squares, to retrieve both parameters a and b optimized for a minimum sum of residuals. If set to *true*, the calculation will keep the offset a fixed. This will force the line to cut the ordinate at the Offset 122 property entered.

Calculation Results (Public Properties)

Delphi: `property Offset: Double read GetOffset write SetOffset;`

C++: `__property double Offset = {read=GetOffset, write=SetOffset};`

If FixedOffset 122 is set to *true*, the property is used to enter the value of the fixed offset a. Otherwise it returns the optimized calculation result for a.

Delphi: `property DeltaOffset: Double read GetDeltaOffset;`

C++: `__property double DeltaOffset = {read=GetDeltaOffset};`

The statistical uncertainty of the offset 122 a.

Delphi: `property Slope: Double read GetSlope;`

C++: `__property double Slope = {read=GetSlope};`

The calculated slope 122 b of the line regression line.

Delphi: `property DeltaSlope: Double read GetDeltaSlope;`
C++: `__property double DeltaSlope = {read=GetDeltaSlope};`
↳ The statistical uncertainty of the slope 122 b.

Delphi: `property AverageX: Double read GetAverageX;`
C++: `__property double AverageX = {read=GetAverageX};`
↳ The arithmetic mean of the X-values.

$$\bar{X} = \frac{\sum X_i}{n}$$

Delphi: `property AverageY: Double read GetAverageY;`
C++: `__property double AverageY = {read=GetAverageY};`
↳ The arithmetic mean of the Y-values.

$$\bar{Y} = \frac{\sum Y_i}{n}$$

Delphi: `property VarianceX: Double read GetVarianceX;`
C++: `__property double VarianceX = {read=GetVarianceX};`
↳ The variance (σ) of the X-values.

$$Q_{XX} = \sum (X_i - \bar{X})^2 \quad \sigma_X^2 = \frac{Q_{XX}}{n-1}$$

Delphi: `property VarianceY: Double read GetVarianceY;`
C++: `__property double VarianceY = {read=GetVarianceY};`
↳ The variance (σ) of the Y-values.

$$Q_{YY} = \sum (Y_i - \bar{Y})^2 \quad \sigma_Y^2 = \frac{Q_{YY}}{n-1}$$

Delphi: `property SumOfResiduals: Double read GetSumOfResiduals;`
C++: `__property double SumOfResiduals = {read=GetSumOfResiduals};`
↳ The sum of the residuals (square deviations) from the calculated line to the measured data points.

Delphi: `property Probability: Double read Getr2;`
C++: `__property double Probability = {read=Getr2};`
↳ The probability (r^2) the linear model fits to the empirical data points.

$$\delta_i = Y_i - f(X_i) \quad Q_{YY} = \sum (Y_i - \bar{Y})^2 \quad r^2 = 1 - \frac{\sum \delta_i^2}{Q_{YY}}$$

Delphi: `property Correlation: Double read GetR;`
C++: `__property double Correlation = {read=GetR};`
↳ The correlation (R) of the linear regression.

$$R = \frac{b\sigma_X}{\sigma_Y}$$

3.5.22 TRtGeneralLinearLeastSquares

This class derived from TRtFunctionSeries 119 is used to display general linear least squares function line through the source points using the basis equations system $Y = \sum_i a_i f_i(X)$.

Unit/Namespace: `RtPolynomial`

Declaration

Delphi:
```
TRtCustomGeneralLinearLeastSquares = class
(TRtFunctionSeries);
TRtGeneralLinearLeastSquares = class
(TRtCustomGeneralLinearLeastSquares);
```
C++:
```
class TRtCustomGeneralLinearLeastSquares : public
Rtfunctionseries::TRtFunctionSeries;
class TRtGeneralLinearLeastSquares : public
TRtCustomGeneralLinearLeastSquares;
```

Published Property

Delphi:
```
property Order: Integer read GetOrder write SetOrder;
```
C++:
```
__property int Order = {read=GetOrder, write=SetOrder,
nodefault};
```
The order of the optimized linear function system.

Calculation Results (Public Properties)

Delphi:
```
property Coefficients[i:Integer]: Double read GetCoeff;
```
C++:
```
__property double Coefficients[int i] = {read=GetCoeff};
```
The coefficients of the optimized linear function system a_i as an array[0..Order] of double.

Calculation Results (Public Properties)

Delphi:
```
property DeltaCoefficients[i:Integer]: Double read
GetDeltaCoeff;
```
C++:
```
__property double DeltaCoefficients[int i] = {read=GetDelta
Coeff};
```
The standard deviations of the coefficients of the optimized linear function system a_i as an array[0..Order] of double.

Delphi:
```
property AverageX: Double read GetAverageX;
```
C++:
```
__property double AverageX = {read=GetAverageX};
```
The arithmetic mean of the X-values.

Delphi:
```
property AverageY: Double read GetAverageY;
```
C++:
```
__property double AverageY = {read=GetAverageY};
```
The arithmetic mean of the Y-values.

Delphi:
```
property VarianceX: Double read GetVarianceX;
```
C++:
```
__property double VarianceX = {read=GetVarianceX};
```
The variance (σ) of the X-values.

Delphi: `property VarianceY: Double read GetVarianceY;`
C++: `__property double VarianceY = {read=GetVarianceY};`
↳ The variance (σ) of the Y-values.

Delphi: `property SumOfResiduals: Double read GetSumOfResiduals;`
C++: `__property double SumOfResiduals = {read=GetSumOfResiduals};`
↳ The sum of the residuals (square deviations) from the calculated line to the measured data points.

Delphi: `property Probability: Double read Getr2;`
C++: `__property double Probability = {read=Getr2};`
↳ The probability (r^2) that the linear model fits to the empirical data points.

Event

Delphi: `property BasisFunction: TRtGeneralLinearFittingHandler` 46 `read GetBaseFn write SetBaseFn;`
C++: `__property TRtFittingFunctionHandler BasisFunction = {read= GetBaseFn, write=SetBaseFn};`
↳ This event is triggered each time a function value of the linear function system is needed to built up the basis function start matrix and for calculating the function line values after optimization. You must use the *Index* supplied and return the function value of the independent variable *X* depending on this matrix column index.
To return a polynomial model fit you would write

```
Result := IntPower(e.X,e.Index);
```

for example

3.5.23 TRtPolynomial Class

This class derived from TRtCustomGeneralLeastSquares 124 is used to display the calculated polynomial regression 47 line $Y = a+bX+cX^2$...

Unit/Namespace: `RtPolynomial`

Declaration

Delphi: `TRtPolynomial = class(TRtCustomGeneralLinearLeastSquares);`
C++: `class TRtPolynomial : public TRtCustomGeneralLinearLeastSquares;`

3.5.24 TRtFittedLine Class

This class derived from TRtFunctionSeries 119 is used to calculate and display the non-linear regression 47 to any function $Y = f(X)$ given the right start values for a, b... The formula can be supplied as method to the FittingFunction 126 event or entered as formula string and solved by a built-in parser.

Unit/Namespace: `RtFitting`

Declaration

Delphi: `TRtFittedLine = class(TRtFunctionSeries);`
C++: `class TRtFittedLine : public Rtfunctionseries::TRtFunctionSeries;`

Published Properties

Delphi: `property Order: Integer read GetOrder write SetOrder;`
C++: `__property int Order = {read=GetOrder, write=SetOrder, nodefault};`

The order of the calculated function.

Delphi: `property Coefficients: TDoubleDynArray read GetCoefficients write SetCoefficients;`
C++: `__property TDoubleDynArray Coefficients = {read=GetCoefficients, write=SetCoefficients};`

The coefficients of the calculated function a, b, c... as an array[0..Order] of double.

Delphi: `property Iterations: Integer read GetIterations write SetIterations;`
C++: `__property int Iterations = {read=GetIterations, write=SetIterations, nodefault};`

The number of iterations used for the optimization of the non-linear function. Setting $Iterations = 0$ will result a function plotter showing the formula using the pre set parameters.

Delphi: `property Expression: string read GetExpression write SetExpression;`
C++: `__property AnsiString Expression = {read=GetExpression, write=SetExpression};`

Fitted expression as string. Only parsed if the FittingFunction 126 is not defined. The parser will also set the correct Order 126 depending on the highest-order coefficient used in the formula.

Delphi: `property DoCovariance: Boolean read GetDoCovariance write SetDoCovariance;`
C++: `__property bool DoCovariance = {read=GetDoCovariance, write=SetDoCovariance, nodefault};`

If set to *false* (normal and recommended case), the fit will optimize the variance to be minimal at the solution. If set to *true*, the covariance will be optimized instead.

Events

Delphi: `property FittingFunction: TRtFittingFunctionHandler 48 read GetFittingFunction write SetFittingFunction;`
C++: `__property TRtFittingFunctionHandler FittingFunction = {read=GetFittingFunction, write=SetFittingFunction};`

This event is triggered each time a function value of the non linear function to fit is needed. You must use the coefficients supplied and return the non linear function value.

Delphi: `property OnNewCorner: FittingCornerHandler 48 read GetOnNewCorner write SetOnNewCorner;`
C++: `__property FittingCornerHandler OnNewCorner = {read=GetOnNewCorner, write=SetOnNewCorner};`

Event triggered each time a new simplex corner (set of coefficients) is generated for testing. Can be used to prepare pre-calculations for your FittingFunction 126, which will be called directly afterwards for each X-value of the data vector supplied.

Delphi: `property OnIteration: TNotifyEvent read GetOnIteration write SetOnIteration;`
C++: `__property Classes::TNotifyEvent OnIteration = {read=GetOnIteration, write=SetOnIteration};`
↳ Triggered after each iteration. Can be used to display iteration results or test if the fit should be continued.

Public Method

Delphi: `procedure StopIterations;`
C++: `void __fastcall StopIterations(void);`
↳ Breaks the optimization iterations loop; the iterations counter is not completed.

Calculation Results (Public Properties)

Delphi: `property AverageX: Double read GetAverageX;`
C++: `__property double AverageX = {read=GetAverageX};`
↳ The arithmetic mean of the X-values.

Delphi: `property AverageY: Double read GetAverageY;`
C++: `__property double AverageY = {read=GetAverageY};`
↳ The arithmetic mean of the Y-values.

Delphi: `property VarianceX: Double read GetVarianceX;`
C++: `__property double VarianceX = {read=GetVarianceX};`
↳ The variance (σ) of the X-values.

Delphi: `property VarianceY: Double read GetVarianceY;`
C++: `__property double VarianceY = {read=GetVarianceY};`
↳ The variance (σ) of the Y-values.

Delphi: `property SumOfResiduals: Double read GetSumOfResiduals;`
C++: `__property double SumOfResiduals = {read=GetSumOfResiduals};`
↳ The sum of the residuals (square deviations) from the calculated line to the measured data points.

Delphi: `property Probability: Double read Getr2;`
C++: `__property double Probability = {read=Getr2};`
↳ The probability (r^2) that the function model fits to the empirical data points.

3.5.25 TRtInterpolation Class

This class derived from TRtFunctionSeries 119 is used to calculate and display an interpolated line 50 between the source data points.

Unit/Namespace: `RtInterpolation`

Declaration

Delphi: `TRtInterpolation = class(TRtFunctionSeries);`
C++: `class TRtInterpolation : public Rtfunctionseries::TRtFunctionSeries;`

Published Properties

Delphi: `property InterpolationMethod: TRtInterpolationMethod read GetInterpolationMethod write SetInterpolationMethod;`
C++: `__property Rtinterpolationcalculations::TRtInterpolationMethod InterpolationMethod = {read=GetInterpolationMethod, write=SetInterpolationMethod, nodefault};`

The method used for interpolation: Akima, Cubic Spline or Approximative Spline.

Delphi: `property SmoothPoints: Boolean read GetDoSmoothing write SetDoSmoothing;`
C++: `__property bool SmoothPoints = {read=GetDoSmoothing, write=SetDoSmoothing, nodefault};`

If set to *true*, then the data points are smoothed using a linearly-weighted moving average (Akima and Cubic Spline only).

Delphi: `property SmoothingRange: Double read GetSmoothM write SetSmoothM;`
C++: `__property double SmoothingRange = {read=GetSmoothM, write=SetSmoothM};`

The range used for the smoothing moving average (Akima and Cubic Spline only).

Delphi: `property ElasticityFactor: Double read GetElasticityFactor write SetElasticityFactor;`
C++: `__property double ElasticityFactor = {read=GetElasticityFactor, write=SetElasticityFactor};`

The weighting factor that determines the effect of the data points to be passed (Approximative Spline only).

Calculation Results (Public Properties)

Delphi: `property Minima: TRtDoublePointArray read GetMinima;`
C++: `__property Rtinterpolationcalculations::TRtDoublePointArray Minima = {read=GetMinima};`

The local minima of the interpolation line.

Delphi: `property Maxima: TRtDoublePointArray read GetMaxima;`
C++: `__property Rtinterpolationcalculations::TRtDoublePointArray Maxima = {read=GetMaxima};`

The local maxima of the interpolation line.

3.5.26 TRtDifferential Class

This class derived from TRtInterpolation is used to calculate and display the differential of an interpolated line between the source data points.

Unit/Namespace: `RtDifferential`

Declaration

Delphi: `TRtDifferential = class(TRtInterpolation);`
C++: `class TRtDifferential : public Rtinterpolation::TRtInterpolation;`

Calculation Results (Public Properties)

Delphi: `property Minima: TRtDoublePointArray` 50 `read GetMinima;`
C++: `__property Rtinterpolationcalculations::TRtDoublePointArray Minima = {read=GetMinima};`

The local minima of the differential of the interpolated line between the source points.

Delphi: `property Maxima: TRtDoublePointArray` 50 `read GetMaxima;`
C++: `__property Rtinterpolationcalculations::TRtDoublePointArray Maxima = {read=GetMaxima};`

The local maxima of the differential of the interpolated line between the source points.

3.5.27 TRtIntegral Class

This class derived from TRtDifferential 128 is used to calculate and display the integral of an interpolated line 50 between the source data points.

Unit/Namespace: `RtIntegral`

Declaration

Delphi: `TRtIntegral = class(TRtDifferential);`
C++: `class TRtIntegral : public Rtdifferential::TRtDifferential;`

Calculation Results (Public Properties)

Delphi: `property Minima: TRtDoublePointArray` 50 `read GetMinima;`
C++: `__property Rtinterpolationcalculations::TRtDoublePointArray Minima = {read=GetMinima};`

The local minima of the integral of the interpolated line between the source points.

Delphi: `property Maxima: TRtDoublePointArray` 50 `read GetMaxima;`
C++: `__property Rtinterpolationcalculations::TRtDoublePointArray Maxima = {read=GetMaxima};`

The local maxima of the integral of the interpolated line between the source points.

3.5.28 TRtMovingAverage Class

This class derived from TRtFunctionSeries 119 is used to calculate and display a moving average 52 line. It is often used in trend analysis in financial charts.

Unit/Namespace: `RtMovingAverage`

Declaration

Delphi: `TRtMovingAverage = class(TRtFunctionSeries);`
C++: `class TRtMovingAverage : public Rtfunctionseries::TRtFunctionSeries;`

Published Properties

Delphi:
```
TRtAveragingMethod=(Simple,LinearWeighted,
ExponentiallyWeighted)
property AveragingMethod: TRtAveragingMethod read
GetAveragingMethod write SetAveragingMethod;
```
C++:
```
enum TRtAveragingMethod { Simple, LinearWeighted,
ExponentiallyWeighted };
__property TRtAveragingMethod AveragingMethod =
{read=GetAveragingMethod, write=SetAveragingMethod,
nodefault};
```
↳ The weighting method used for the sum: simple, linear or exponential.

Delphi:
```
TRtAveragingDirection=(OnlyDownwards,UpAndDownwards);
property AveragingDirection: TRtAveragingDirection read
GetAveragingDirection write SetAveragingDirection;
```
C++:
```
enum TRtAveragingDirection { OnlyDownwards, UpAndDownwards };
__property TRtAveragingDirection AveragingDirection =
{read=GetAveragingDirection, write=SetAveragingDirection,
nodefault};
```
↳ Specifies whether the sum is taken only upwards or upwards and downwards to the point of interest.

Delphi:
```
TRtAveragingRangeMethod=(NasCount,NasXDistance);
property AveragingRangeMethod: TRtAveragingRangeMethod read
GetAveragingRangeMethod write SetAveragingRangeMethod;
```
C++:
```
enum TRtAveragingRangeMethod { NasCount, NasXDistance };
__property TRtAveragingRangeMethod AveragingRangeMethod =
{read=GetAveragingRangeMethod, write=SetAveragingRangeMethod,
nodefault};
```
↳ Specifies whether the range for the sum (N 130) is interpreted as a count or as an X-distance.

Delphi:
```
property N: Double read GetN write SetN;
```
C++:
```
__property double N = {read=GetN, write=SetN};
```
↳ The averaging range/count N used for the averaging calculation.

Delphi:
```
property ExponentialFactorForN: Double read
GetExponentialFactorForN write SetExponentialFactorForN;
```
C++:
```
__property double ExponentialFactorForN =
{read=GetExponentialFactorForN,
write=SetExponentialFactorForN};
```
↳ The factor used with exponential weighted averaging method 130.

3.5.29 TRtDiscreteFourier Class

This class derived from TRtFunctionSeries 119 is used to display the frequency spectrum of a source data set. It is best performing if the source dada are equally spaced, ascending in X and the count is a power of 2.

Unit/Namespace: `RtFFT`

Declaration

Delphi:
```
TRtDiscreteFourier = class(TRtFunctionSeries);
```
C++:
```
class TRtDiscreteFourier : public TRtFunctionSeries;
```

Delphi: `property Phase: TRtChildSeries read FPhaseSeries write FPhaseSeries;`
C++: `__property TRtChildSeries* Phase = {read=FPhaseSeries, write=FPhaseSeries};`
The phase of the complex frequency value factors can be displayed in an optional point series.

3.5.30 TRtCustomLegend Class

This class provides properties and methods common to the legend for the Cartesian graph control and the legend for the pie/donut chart control.

Unit/Namespace: `RtLegend`

Declaration

Delphi: `TRtCustomLegend = class(TGraphicControl);`
C++: `class TRtCustomLegend : public Controls::TGraphicControl;`

Published Properties

Delphi: `property Caption: TRtRichCaption read GetCaption write SetCaption;`
C++: `__property WideString Caption = {read=GetCaption, write=SetCaption};`
The optional caption displayed on top of the legend.

Delphi: `property CaptionVisible: Boolean read FCaptionVisible write SetCaptionVisible;`
C++: `__property bool CaptionVisible = {read=FCaptionVisible, write=SetCaptionVisible, nodefault};`
If set to *true*, the caption will be displayed on top of the legend.

Delphi: `property CaptionAlignment: TAlignment read FCaptionAlignment write SetCaptionAlignment;`
C++: `property Classes::TAlignment CaptionAlignment = {read=FCaptionAlignment, write=SetCaptionAlignment, nodefault};`
The alignment of the title caption relative to the control width.

Delphi: `property CaptionFont: TRtFont read GetCaptionFont write SetCaptionFont;`
C++: `__property Rtrichlabel::TRtFont* CaptionFont = {read=GetCaptionFont, write=SetCaptionFont};`
The font with enhanced settings used for the title caption.

Delphi: `property CaptionColor: TRtColor read GetCaptionColor write SetCaptionColor;`
C++: `__property Rtgdi::TRtColor CaptionColor = {read=GetCaptionColor, write=SetCaptionColor, nodefault};`
The color of the title caption.

Delphi: `property CaptionDistanceBefore: Single read FCaptionDistanceBefore write SetCaptionDistanceBefore;`
C++: `__property float CaptionDistanceBefore = {read=FCaptionDistanceBefore, write=SetCaptionDistanceBefore};`
The distance of the title caption to the top border of the control.

Delphi: `property CaptionDistanceAfter: Single read FCaptionDistanceAfter write SetCaptionDistanceAfter;`
C++: `__property float CaptionDistanceAfter = {read=FCaptionDistanceAfter, write=SetCaptionDistanceAfter};`
The distance from the lower edge of the title caption to the items of the legend.

Delphi: `property BackColor: TRtColor read FBackColor write SetBackColor;`
C++: `__property Rtgdi::TRtColor BackColor = {read=FBackColor, write=SetBackColor, nodefault};`
The background color of the control.

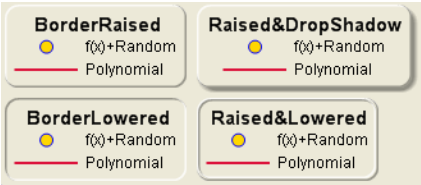
Delphi: `property BackgroundGradient: TRtGradientSettings read FBackgroundGradient write FBackgroundGradient;`
C++: `__property TRtGradientSettings* BackgroundGradient = {read=FBackgroundGradient, write=FBackgroundGradient};`
Settings for an optional gradient drawn at the background of the legend.

Delphi: `property ForeColor: TRtColor read FForeColor write SetForeColor;`
C++: `__property Rtgdi::TRtColor ForeColor = {read=FForeColor, write=SetForeColor, nodefault};`
The color used for the foreground (the items captions) in the control.

Delphi: `property Font: TRtFont read FFont write SetFont stored GetStoreFont;`
C++: `__property Rtrichlabel::TRtFont* Font = {read=FFont, write=SetFont, stored=GetStoreFont};`
The font with enhanced settings used for the items captions.

Delphi: `property Frame: TRtLineSettings read FFrame write FFrame;`
C++: `__property Rtstyles::TRtLineSettings* Frame = {read=FFrame, write=FFrame};`
The settings for the frame line used for the border of the control.

Delphi: `property BorderRaised: Boolean read FBorderRaised write SetBorderRaised;`
C++: `__property bool BorderRaised = {read=FBorderRaised, write=BorderRaised, nodefault};`
If set *true* the legend is drawn with lights and shadows to be raised.



Delphi: `property BorderLowered: Boolean read FBorderLowered write SetBorderLowered;`
C++: `__property bool BorderLowered = {read=FBorderLowered, write=SetBorderLowered, nodefault};`
If set *true* the legend is drawn with lights and shadows to be lowered.

Delphi: `property DropShadow: Boolean read FDropShadow write SetDropShadow;`
C++: `__property bool DropShadow = {read=FDropShadow, write=SetDropShadow, nodefault};`
If set *true* a drop shadow is drawn below the legend on the graph.

Delphi: `TRtItemsOrder=(ByColumn,ByRow);`
`property ItemsOrder: TRtItemsOrder read FItemsOrder write SetItemsOrder;`
C++: `enum TRtItemsOrder { ByColumn, ByRow };`
`__property TRtItemsOrder ItemsOrder = {read=FItemsOrder, write=SetItemsOrder, nodefault};`
The order in which the items are drawn to the legend.

Delphi: `property Rows: Integer read FRows write SetRows;`
C++: `__property int Rows = {read=FRows, write=SetRows, nodefault};`
The number of rows to display. Only used if ItemsOrder 133 is *ByRow*.

Delphi: `property Columns: Integer read FColumns write SetColumns;`
C++: `__property int Columns = {read=FColumns, write=SetColumns, nodefault};`
The number of columns to display. Only used if ItemsOrder 133 is ByColumn.

Delphi: `property ItemHeight: Single read FItemHeightSet write SetItemHeight;`
C++: `__property float ItemHeight = {read=FItemHeightSet, write=SetItemHeight};`
The height of the items displayed.

Delphi: `property ItemWidth: Single read FItemWidth write SetItemWidth;`
C++: `__property float ItemWidth = {read=FItemWidth, write=SetItemWidth};`
The width of the items displayed.

Delphi: `property DistanceItemCaption: Single read FDistanceItemCaption write SetDistanceItemCaption;`
C++: `__property float DistanceItemCaption = {read=FDistanceItemCaption, write=SetDistanceItemCaption};`
The distance between the series item and the corresponding caption to its right.

Delphi: `property DistanceBetweenLines: Single read FDistanceBetweenLines write SetDistanceBetweenLines;`
C++: `__property float DistanceBetweenLines = {read=FDistanceBetweenLines, write=SetDistanceBetweenLines};`
The distance between the item lines if more than one row is displayed.

Delphi: `property DistanceBetweenColumns: Single read FDistanceBetweenColumns write SetDistanceBetweenColumns;`
C++: `__property float DistanceBetweenColumns = {read=FDistanceBetweenColumns, write=SetDistanceBetweenColumns};`
The distance between the item columns if more than one column is displayed.

Delphi: `property DistanceToBorder: Single read FDistanceToBorder write SetDistanceToBorder;`
C++: `__property float DistanceToBorder = {read=FDistanceToBorder, write=SetDistanceToBorder};`
The distance between the items and captions and the control border.

Delphi: `property CornerRounding: Single read FCornerRounding write SetCornerRounding;`
C++: `__property float CornerRounding = {read=FCornerRounding, write=SetCornerRounding};`
The radius used to draw the rounded corners of the control.

Public Methods

Delphi:
```
procedure ReadFrom(FileName: string); overload;
procedure ReadFrom(Stream: TStream); overload; virtual;
procedure ReadFrom(Node: IXMLNode); overload; virtual;
procedure ReadFrom(Ini: TCustomIniFile; Section: string);
overload; virtual;
```
C++:
```
void __fastcall ReadFrom(AnsiString FileName);
virtual void __fastcall ReadFrom(Classes::TStream* Stream);
virtual void __fastcall ReadFrom(Xmlintf::_di_IXMLNode Node);
virtual void __fastcall ReadFrom(Inifiles::TCustomIniFile*
Ini, AnsiString Section);
```
- The legend settings can be read from a file in four different ways:
  from a binary file specified by name,
  from any binary stream,
  from an XML document node,
  from a specified section of an INI file.

Delphi:
```
procedure WriteTo(FileName: string); overload;
procedure WriteTo(Stream: TStream); overload; virtual;
procedure WriteTo(Node: IXMLNode); overload; virtual;
procedure WriteTo(Ini: TCustomIniFile; Section: string);
overload; virtual;
```
C++:
```
void __fastcall WriteTo(AnsiString FileName);
virtual void __fastcall WriteTo(Classes::TStream* Stream);
virtual void __fastcall WriteTo(Xmlintf::_di_IXMLNode Node);
virtual void __fastcall WriteTo(Inifiles::TCustomIniFile*
Ini, AnsiString Section);
```
- The legend settings can be saved to a file in four different ways:
  to a binary file specified by name,
  to any binary stream,
  to an XML document node,
  to a specified section of an INI-file.

Delphi:
```
procedure Modified; virtual;
```
C++:
```
virtual void __fastcall Modified(void);
```
- Marks the fact that the control needs to recalculate its item positions and has to be redrawn (for internal use only).

Events

Delphi:
```
TRtLegendItemCaptionEventArgs=record
      Caption: TRtRichCaption;
      Index: Integer;
end;
TRtLegendItemCaptionEventHandler=procedure(Sender: TObject;
var e: TRtLegendItemCaptionEventArgs) of object;
property OnGetLegendItemCaption:
TRtLegendItemCaptionEventHandler read mGetItemCaption write
mGetItemCaption;
```

C++:
```
struct TRtLegendItemCaptionEventArgs{
public:
      WideString Caption;
      int Index;
};
typedef void __fastcall (__closure *
TRtLegendItemCaptionEventHandler)(System::TObject* Sender,
const TRtLegendItemCaptionEventArgs &e);
__property TRtLegendItemCaptionEventHandler
OnGetLegendItemCaption = {read=mGetItemCaption, write=
mGetItemCaption};
```

- The event is triggered each time a legend item needs to get its related caption. You can use it to perform your own custom drawing of the relevant series captions giving the pre set. You can alter the ***e.Caption*** to your needs.

Delphi:
```
TRtLegendItemDrawArgs = record
  Graphics: TRtGraphics;
  Rect: TRtRectangle;
  Index: Integer;
end;
TRtLegendItemDrawHandler = procedure(Sender: TObject; e:
TRtLegendItemDrawArgs) of object;
property OnDrawLegendItem: TRtLegendItemDrawHandler read
mDrawLegendItem write mDrawLegendItem;
```

C++:
```
struct TRtLegendItemDrawArgs{
public:
      Gdipobj::TGPGraphics* Graphics;
      Gdipapi::TGPRect Rect;
      int Index;
};
typedef void __fastcall (__closure *TRtLegendItemDrawHandler)
(System::TObject* Sender, const TRtLegendItemDrawArgs &e);
__property TRtLegendItemDrawHandler OnDrawLegendItem =
{read=mDrawLegendItem, write=mDrawLegendItem};
```

- The event is triggered each time a legend item needs to be drawn.
  With a **TRtLegend** you can use it to perform your own custom drawing of the relevant series supplied as ***Sender*** to the ***Graphics***, either by using the supplied ***Rect*** or by using the ***Index*** to the series list of the graph.
  With a **TRtPieLegend** however the ***Sender*** is set to the calling legend item **TRtFillSettings** record and ***Index*** is set to the segment item index.

3.5.31 TRtLegend Class

This class provides a component to display a list of all series 26 contained in an assigned graph. It shows all lines and point styles and the corresponding series captions. Note, that the visibilities 105, styles and captions 104 can only be set within each single series component.

Unit/Namespace: `RtLegend`

Declaration

Delphi: `TRtLegend = class(TRtCustomLegend 131);`

C++: `class TRtLegend : public Rtlegend::TRtCustomLegend`

Public Property

Delphi:
```
TRtLegendCaptionColorFrom=(LegendForeColor,SeriesLineColor,
SeriesAreaForeColor);
property LegendCaptionColorFrom: TRtLegendCaptionColorFrom
read FLegendCaptionColorFrom write FLegendCaptionColorFrom;
```
C++:
```
enum TRtLegendCaptionColorFrom { LegendForeColor,
SeriesLineColor, SeriesAreaForeColor };
__property TRtLegendCaptionColorFrom LegendCaptionColorFrom =
{read=FLegendCaptionColorFrom, write=FLegendCaptionColorFrom,
nodefault};
```

Specifies whether the drawing color of single series item captions in the legend is defined by the foreground color of the legend, the line color of the series or the foreground color of the series area.

Published Properties

Delphi:
```
property Graph: TRtGraph2D read FGraph write SetGraph;
```
C++:
```
__property Rtgraph2d::TRtGraph2D* Graph = {read=FGraph,
write=SetGraph};
```

The assigned graph containing the series items to display. At design time it is set automaticallyto the first graph 23 you place the control to.

Delphi:
```
property ItemsOrderList: TStrings read FItemsOrderList write
SetItemsOrderList;
```
C++:
```
__property Classes::TStrings* ItemsOrderList =
{read=FItemsOrderList, write=SetItemsOrderList};
```

If you alter the display order 36 of the series items, this list stores the references to the fully-qualified names of the series in the order of their appearance.

Delphi:
```
TRtLegendPosition=(BottomCenterOfGraph,BottomCenterOfAxis,
LeftCenterOfGraph,LeftCenterOfAxis,LeftTopOfAxis,
TopCenterOfGraph,TopCenterOfAxis,RightCenterOfGraph,
RightCenterOfAxis,RightTopOfAxis,FreeFloating);
property Position: TRtLegendPosition read FPosition write
SetPosition;
```
C++:
```
enum TRtLegendPosition { BottomCenterOfGraph,
BottomCenterOfAxis, LeftCenterOfGraph, LeftCenterOfAxis,
LeftTopOfAxis, TopCenterOfGraph, TopCenterOfAxis,
RightCenterOfGraph, RightCenterOfAxis, RightTopOfAxis,
FreeFloating };
__property TRtLegendPosition Position = {read=FPosition,
write=SetPosition, nodefault};
```

The position of the legend: relative to the graph, relative to the axes or free-floating.

Delphi:
```
property StartFreeFloatingMovingKeys: TShiftState read
FStartMoveKeys write FStartMoveKeys;
```
C++:
```
__property Classes::TShiftState StartFreeFloatingMovingKeys =
{read=FStartMoveKeys, write=FStartMoveKeys, nodefault};
```

The mouse button control key combination used to start the drag operation for the control, if Position 136 is *FreeFloating*.

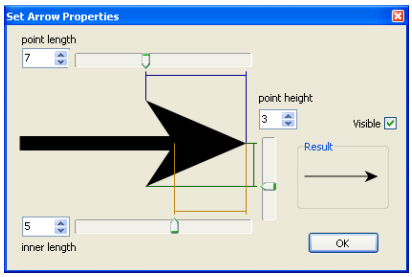
Events

Delphi: `property OnStartFreeFloating: TNotifyEvent read mStartFreeFloating write mStartFreeFloating;`

C++: `__property Classes::TNotifyEvent OnStartFreeFloating = {read=mStartFreeFloating, write=mStartFreeFloating};`

This event is trigered if Position 136 is *FreeFloating* and the user starts the drag operation for the control. It can be used to store undo information at the start of the move operation.

3.5.32 TRtAxisArrowPoint Class



This class is used to control the properties of the optional arrow point 142 at the end of the axis. The styles can be set interactively using a special property editor.

Unit/Namespace: `RtStyles`

Declaration

Delphi: `TRtAxisArrowPoint = class(TPersistent);`

C++: `class TRtAxisArrowPoint : public Classes::TPersistent;`

Published Properties

Delphi: `property Visible: Boolean read FVisible write SetVisible;`

C++: `__property bool Visible = {read=FVisible, write=SetVisible, nodefault};`

Switches the display of the arrow point at the end of the axis on or off.

Delphi: `property PointLength: Single read FPointLength write SetPointLength;`

C++: `__property float PointLength = {read=FPointLength, write=SetPointLength};`

The length of the arrow point as a ratio to axis line thickness 141.

Delphi: `property InnerLength: Single read FInnerLength write SetInnerLength;`

C++: `__property float InnerLength = {read=FInnerLength, write=SetInnerLength};`

The length of the inner part of the arrow point as a ratio to axis line thickness 141.

Delphi: `property PointHeight: Single read FPointHeight write SetPointHeight;`

C++: `__property float PointHeight = {read=FPointHeight, write=SetPointHeight};`

The height of the arrow point as a ratio to axis line thickness 141.

3.5.33 TRtAxis Class

This class provides a component to display an axis. It will automatically adjust itself to other axes and legends contained in the same graph control.

Unit/Namespace: `RtAxis`

Declaration

Delphi: `TRtAxis = class(TGraphicControl;`
C++: `class TRtAxis : public Controls::TGraphicControl;`

Public Properties

Delphi: `property AutoSize: Boolean read FAutoSize write SetAutoSize;`
C++: `__property bool AutoSize = {read=FAutoSize, write=SetAutoSize, nodefault};`
- Automates the sizing of the control and the adjustment of the axis in relation to other axes in the graph. Highly recommended to keep this set to *true*.

Delphi: `property AxLength: Single read FAxLength;`
C++: `__property float AxLength = {read=FAxLength};`
- The length of the axis line.

Delphi: `property AxLeft: Single read FAxLeft;`
C++: `__property float AxLeft = {read=FAxLeft};`
- The internal offset of the axis line from the left of the control.

Delphi: `property AxTop: Single read FAxTop;`
C++: `__property float AxTop = {read=FAxTop};`
- The internal offset of the axis line from the top of the control.

Delphi: `property HasLogScaling: Boolean read FHasLogScaling;`
C++: `__property bool HasLogScaling = {read=FHasLogScaling, nodefault};`
- Determines whether the actual scaling 144 shown is logarithmic.

Delphi: `property NormalizeConst: Double read FNormalizeConst;`
C++: `__property double NormalizeConst = {read=FNormalizeConst};`
- The divisor to normalize the numerical data to graph world positions (for internal use).

Delphi: `property RangeExponent: Integer read FRangeExponent;`
C++: `__property int RangeExponent = {read=FRangeExponent, nodefault};`
- The 10th logarithm of the current start/stop range (for internal use).

Delphi: `property ValuesStart: Double read FValuesStart;`
C++: `__property double ValuesStart = {read=FValuesStart};`
- The start of the values range from the data vectors of the series referring to the axis.

Delphi: `property ValuesStop: Double read FValuesStop;`
C++: `__property double ValuesStop = {read=FValuesStop};`
- The stop of the values range from the data vectors of the series referring to the axis.

Delphi: `property ScrollStart: Double read GetScrollStart;`
C++: `__property double ScrollStart = {read=GetScrollStart};`
↳ The start of the range used to display scrollbars (for internal use).

Delphi: `property ScrollStop: Double read GetScrollStop;`
C++: `__property double ScrollStop = {read=GetScrollStop};`
↳ The stop of the range used to display scrollbars (for internal use).

Delphi: `property ShortMonthFormat: string read FShortMonthFormat write FShortMonthFormat;`
C++: `__property AnsiString ShortMonthFormat = {read=FShortMonthFormat, write=FShortMonthFormat};`
↳ This format string is used if the scaling 144 has been set to *DateTime* and the current range gives an optimally scaled display of labels when setting parts at month intervals. The syntax is the same as in the ***FormatDateTime*** function.

Delphi: `property LongMonthFormat: string read FLongMonthFormat write FLongMonthFormat;`
C++: `__property AnsiString LongMonthFormat = {read=FLongMonthFormat, write=FLongMonthFormat};`
↳ This format string is used if the scaling 144 has been set to *DateTime* and the current range gives an optimally scaled display of labels when setting parts at month intervals.

Delphi: `property DayFormat: string read DayFormat write DayFormat;`
C++: `__property AnsiString DayFormat = {read=FDayFormat, write=FDayFormat};`
↳ This format string is used if the scaling 144 has been set to *DateTime* and the current range gives an optimally scaled display of labels when setting parts at day intervals.

Delphi: `property DayAndHourFormat: string read FDayAndHourFormat write FDayAndHourFormat;`
C++: `__property AnsiString DayAndHourFormat = {read=FDayAndHourFormat, write=FDayAndHourFormat};`
↳ This format string is used if the scaling 144 has been set to *DateTime* and the current range gives an optimally scaled display of labels when setting parts at hour intervals.

Delphi: `property MinuteFormat: string read FMinuteFormat write FMinuteFormat;`
C++: `__property AnsiString MinuteFormat = {read=FMinuteFormat, write=FMinuteFormat};`
↳ This format string is used if the scaling 144 has been set to *DateTime* and the current range gives an optimally scaled display of labels when setting parts at minute intervals.

Delphi: `property SecondFormat: string read FSSecondFormat write FSecondFormat;`
C++: `__property AnsiString SecondFormat = {read=FSecondFormat, write=FSecondFormat};`
↳ This format string is used if the scaling 144 has been set to *DateTime* and the current range gives an optimally scaled display of labels when setting parts at second intervals.

Delphi: `property MilliSecondFormat: string read FMilliSecondFormat write FMilliSecondFormat;`
C++: `__property AnsiString MilliSecondFormat = {read=FMilliSecondFormat, write=FMilliSecondFormat};`

↳ This format string is used if the scaling 144 has been set to *DateTime* and the current range gives an optimally scaled display of labels when setting parts at millisecond intervals.

Published Properties

Delphi: `property Caption: TRtRichCaption 70 read GetCaption write SetCaption;`
C++: `__property WideString Caption = {read=GetCaption, write=SetCaption};`

↳ The caption displayed at the axis.

Delphi: `property CaptionVisible: Boolean read FCaptionVisible write SetCaptionVisible;`
C++: `__property bool CaptionVisible = {read=FCaptionVisible, write=SetCaptionVisible, nodefault};`

↳ If set to *true*, the caption will be displayed with the axis.

Delphi: `property CaptionAlignment: TAlignment read FCaptionAlignment write SetCaptionAlignment;`
C++: `property Classes::TAlignment CaptionAlignment = {read=FCaptionAlignment, write=SetCaptionAlignment, nodefault};`

↳ The alignment of the title caption relative to the control width/height.

Delphi: `property CaptionHorizontalOnVerticalAxis: Boolean read FHorizontalCaptiononVerticalAxis write SetHorizontalCaptiononVerticalAxis;`
C++: `__property bool CaptionHorizontalOnVerticalAxis = {read=FHorizontalCaptionOnVerticalAxis, write=SetHorizontalCaptiononVerticalAxis, nodefault};`

↳ Set to *true* if you want to draw the caption horizontally on vertical (Y) axes.

Delphi: `property CaptionFont: TRtFont 66 read GetCaptionFont write SetCaptionFont;`
C++: `__property Rtrichlabel::TRtFont* CaptionFont = {read=GetCaptionFont, write=SetCaptionFont};`

↳ The font with enhanced settings used for the axis caption 131.

Delphi: `property CaptionColor: TRtColor 58 read GetCaptionColor write SetCaptionColor;`
C++: `__property Rtgdi::TRtColor CaptionColor = {read=GetCaptionColor, write=SetCaptionColor, nodefault};`

↳ The color of the axis caption.

Delphi: `property CaptionDistanceBefore: Single read FCaptionDistanceBefore write SetCaptionDistanceBefore;`
C++: `__property float CaptionDistanceBefore = {read=FCaptionDistanceBefore, write=SetCaptionDistanceBefore};`

↳ The distance of the caption to the axis numbers for bottom axes, to the top border of the control for top axes, to the left of the control for left axes and to the axis numbers for right axes.

Delphi: `property CaptionDistanceAfter: Single read FCaptionDistanceAfter write SetCaptionDistanceAfter;`
C++: `__property float CaptionDistanceAfter = {read=FCaptionDistanceAfter, write=SetCaptionDistanceAfter};`
↳ The distance of the caption to the axis numbers for top axes, to the top border of the control for bottom axes, to the left of the control for right axes and to the axis numbers for left axes.

Delphi: `property NumbersVisible: Boolean read FNumbersVisible write SetNumbersVisible;`
C++: `__property bool NumbersVisible = {read=FNumbersVisible, write=SetNumbersVisible, nodefault};`
↳ Set to *true* to show the scaling number labels.

Delphi: `property Font: TRtFont` 66 `read FFont write SetFont stored GetStoreFont;`
C++: `__property Rtrichlabel::TRtFont* Font = {read=FFont, write=SetFont, stored=GetStoreFont};`
↳ The font with enhanced settings used for the scaling number labels.

Delphi: `property RubricsAngle: Single read FRubricsAngle write SetRubricsAngle;`
C++: `__property float RubricsAngle = {read=FRubricsAngle, write=SetRubricsAngle};`
↳ The angle in degrees at which the scaling number labels are to be drawn.

Delphi: `property NumbersColor: TRtColor` 58 `read FNumbersColor write SetNumbersColor;`
C++: `__property Rtgdi::TRtColor NumbersColor = {read=FNumbersColor, write=SetNumbersColor, nodefault};`
↳ The color of the scaling number labels.

Delphi: `property ExtraDigits: Integer read FExtraDigits write SetExtraDigits;`
C++: `__property int ExtraDigits = {read=FExtraDigits, write=SetExtraDigits, nodefault};`
↳ The scaling number labels are normally optimized to have the lowest possible number of digits. You can increase the number of digits using this property. Setting ***ExtraDigits*** to 1 will therefore display 3.0 instead of 3.

Delphi: `property ForeColor: TRtColor` 58 `read FForeColor write SetForeColor;`
C++: `__property Rtgdi::TRtColor ForeColor = {read=FForeColor, write=SetForeColor, nodefault};`
↳ The color of the axis line. If you change this color and the TickColor 142 or MajorTickColor 142 are equal to the foreground color then they will also be changed.

Delphi: `property AxisThicknes: Single read FAxisThick write SetAxisThicknes;`
C++: `__property float AxisThicknes = {read=FAxisThick, write=SetAxisThicknes};`
↳ The width of the axis line.

Delphi: `property TicksPointToData: Boolean read FTicksPointToData write SetTicksPointToData;`
C++: `__property bool TicksPointToData = {read=FTicksPointToData, write=SetTicksPointToData, nodefault};`
↳ The direction in which the scaling ticks are drawn in relation to the graph data area.

Delphi: `property TickThicknes: Single read FTickThick write SetMinorTickThicknes;`
C++: `__property float TickThicknes = {read=FTickThick, write=SetMinorTickThicknes};`
The line width of the minor scaling ticks.

Delphi: `property TickLength: Single read FTickLength write SetTickLength;`
C++: `__property float TickLength = {read=FTickLength, write=SetTickLength};`
The length of the minor scaling ticks.

Delphi: `property TickColor: TRtColor read FTickColor write SetTickColor;`
C++: `__property Rtgdi::TRtColor TickColor = {read=FTickColor, write=SetTickColor, nodefault};`
The color of the minor scaling ticks.

Delphi: `property MajorTickThicknes: Single read FMajorTickThick write SetMajorTickThicknes;`
C++: `__property float MajorTickThicknes = {read=FMajorTickThick, write=SetMajorTickThicknes};`
The line width of the major scaling ticks.

Delphi: `property MajorTickLength: Single read FMajorTickLength write SetMajorTickLength;`
C++: `__property float MajorTickLength = {read=FMajorTickLength, write=SetMajorTickLength};`
The length of the major scaling ticks.

Delphi: `property MajorTickColor: TRtColor read FMajorTickColor write SetMajorTickColor;`
C++: `__property Rtgdi::TRtColor MajorTickColor = {read=FMajorTickColor, write=SetMajorTickColor, nodefault};`
The color of the major scaling ticks.

Delphi: `property ArrowPoint: TRtAxisArrowPoint read FArrowPoint write FArrowPoint;`
C++: `__property Rtstyles::TRtAxisArrowPoint* ArrowPoint = {read=FArrowPoint, write=FArrowPoint};`
The axis can be drawn with an arrow point at the end. This property controls its style.

Delphi:
```
TRtAxisPosition = (Bottom, Left, Top, Right);
property Position: TRtAxisPosition read FPosition write SetPosition;
```
C++:
```
enum TRtAxisPosition { Bottom, Left, Top, Right };
__property TRtAxisPosition Position = {read=FPosition, write=SetPosition, nodefault};
```
The position of the axis in the graph. Note that you can create as many axes you need at any position. They will arrange themselves automatically.

Delphi: `property Start: Double read FStart write SetStart;`
C++: `__property double Start = {read=FStart, write=SetStart};`
The start of the axis range. Note that this may change automatically, depending on the StartEnd and the series data of the graph.

Delphi: `property Stop: Double read FStop write SetStop;`
C++: `__property double Stop = {read=FStop, write=SetStop};`

The stop of the axis range. Note that this may change automatically, depending on the StopEnd 143 and the series data of the graph.

Delphi:
```
TRtAxisEndRounding = (RndToMajorZoomToMinor,
RndToMajorZoomToNothing, RndToMajor, RndToMinorZoomToNothing,
RndToMinor, RndNothing);
property EndRounding: TRtAxisEndRounding read FEndRounding
write SetEndRounding;
```
C++:
```
enum TRtAxisEndRounding { RndToMajorZoomToMinor,
RndToMajorZoomToNothing, RndToMajor, RndToMinorZoomToNothing,
RndToMinor, RndNothing };
__property TRtAxisEndRounding EndRounding =
{read=FEndRounding, write=SetEndRounding, nodefault};
```

The ends of the axis are rounded to the relevant ticks:

RndToMajorZoomToMinor: Rounds to the next major tick if all values are shown. If zoomed, round to the next minor ticks.
RndToMajorZoomToNothing: Rounds to the next major tick if all values are shown. If zoomed, nothing is rounded.
RndToMajor: Rounds to major ticks even when zoomed.
RndToMinorZoomToNothing: Rounds to the next minor tick if all values are shown. If zoomed, nothing is rounded.
RndToMinor: Rounds to minor ticks even when zoomed.
RndNothing: Nothing is rounded.

Delphi:
```
TRtAxisEnd = (AutoRange, FirstSet, Fixed, MovingSlit);
property StartEnd: TRtAxisEnd read FStartEnd write
SetStartEnd;
```
C++:
```
enum TRtAxisEnd { AutoRange, FirstSet, Fixed, MovingSlit };
__property TRtAxisEnd StartEnd = {read=FStartEnd,
write=SetStartEnd, nodefault};
```

When using graph series with AutoUpdate 104 set to *true*, the values range will be updated with the actual range of all series to display. If *AutoRange* is set, the axis will adapt its range to display all series values. If set to *FirstSet*, the start will only be updated if the series ValuesStart 138 becomes lower than the value contained in the Start 142 property. This option is useful for an online data acquisition display, which should have a fixed range scale to start with and which should increase its scale if acquired values become lower than the initial values. Setting this to *Fixed* will switch off auto ranging and always display the axis with its initial start value.

Delphi: `property StopEnd: TRtAxisEnd read FStopEnd write SetStopEnd;`
C++: `__property TRtAxisEnd StopEnd = {read=FStopEnd, write=SetStopEnd, nodefault};`

- When using graph series with AutoUpdate 104 set to *true*, the values range will be updated with the actual range of all series to display. If *AutoRange* is set, the axis will adapt its range to display all series values. If set to *FirstSet*, the start will only be updated if the series ValuesStop 138 becomes greater than the value contained in the Stop 143 property. This option is useful for an online data acquisition display, which should have a fixed range scale to start with and which should increase its scale if acquired values become greater than the initial values. Setting this to *Fixed* will switch off auto ranging and always display the axis with its initial stop value. Setting this to *MovingSlit* yields a special "moving window" ranging. The previously set distance (MovingSlitWidth 145) to the start will be kept constant and moved by the MovingSlitIncrement 145. This option is useful for example for an online data display, which should always display a constant range, such as the the last minute of acquisition, for example.

Delphi: `property Twisted: Boolean read FTwisted write SetTwisted;`
C++: `__property bool Twisted = {read=FTwisted, write=SetTwisted, nodefault};`

- If set to *true*, the display of the numbers will be twisted. The smaller values will be more to the right/top. This setting makes sense when plotting infrared spectra using wave numbers as units, for example.

Delphi:
```
TRtAxisScaling = (Normal, Log, LogEE, Seconds, DateTime, Rubrics);
property Scaling: TRtAxisScaling read FScaling write SetScaling;
```
C++:
```
enum TRtAxisScaling { Normal, Log, LogEE, Seconds, DateTime, Rubrics };
__property TRtAxisScaling Scaling = {read=FScaling, write=SetScaling, nodefault};
```

- The type of scaling: normal, logarithmic, logarithmic with exponential numbers, expired time in seconds, date/time or fixed rubrics.

Delphi: `property MinForExtraExponent: Double read FMinForExtraExponent write SetMinForExtraExponent;`
C++: `__property double MaxForExtraExponent = {read=FMinForExtraExponent, write=SetMinForExtraExponent};`

- If the axis scaling has been set to *Normal*, the number labels can be displayed with an additional exponent label giving the decimal values range. If the values range is lower than this property, this special display will be activated.

Delphi: `property MaxForExtraExponent: Double read FMaxForExtraExponent write SetMaxForExtraExponent;`
C++: `__property double MaxForExtraExponent = {read=FMaxForExtraExponent, write=SetMaxForExtraExponent};`

- If the axis scaling has been set to *Normal*, the number labels can be displayed with an additional exponent label giving the decimal values range. If the values exceed this property, this special display will be activated.

Delphi: `property LogToNormalScalingLevel: Double read FLogToNormalScalingLevel write SetLogToNormalScalingLevel;`
C++: `__property double LogToNormalScalingLevel = {read=FLogToNormalScalingLevel, write=SetLogToNormalScalingLevel};`

This property applies when scaling 144 is set to *Log* or *LogEE* only. Logarithmic scaling is especially useful for displaying data with very high ranges. If you successively zoom into such a log scale, there is a point beyond which the scaling ceases to display any useful scaling numbers or major ticks. You may therefore want to switch once more to *Normal* scaling. This behavior can be activated by setting a non-zero value for this property. If the range of a log axis becomes lower than this level, the scaling displayed will show normal linear partitions. It will be restored to logarithmic scaling again when you zoom back out beyond the range given.

Delphi: `property MovingSlitWidth: Double read FMovingSlitWidth write SetMovingSlitWidth;`
C++: `__property double MovingSlitWidth = {read=FMovingSlitWidth, write=SetMovingSlitWidth};`

This property applies to scaling 144 set to *MovingSlit* only. This is the range the "moving window" will show up to the last acquired value.

Delphi: `property MovingSlitIncrement: Double read FMovingSlitIncrement write SetMovingSlitIncrement;`
C++: `__property double MovingSlitIncrement = {read=FMovingSlitIncrement, write=SetMovingSlitIncrement};`

This property applies to scaling 144 set to *MovingSlit* only. This is the increment the "moving window" will shift by if the last acquired value no longer fits the range.

Delphi: `property Rubrics: TRtCaptions 112 read GetRubrics write SetRubrics;`
C++: `__property Tntclasses::TTntStrings* Rubrics = {read=GetRubrics, write=SetRubrics};`

This property applies to scaling 144 set to *Rubrics* only. This is the list of captions used as labels for the rubrics at the axis.

Delphi: `property DataSource: TDataSource read GetDataSource write SetDataSource;`
C++: `__property Db::TDataSource* DataSource = {read=GetDataSource, write=SetDataSource};`

This property applies to scaling 144 set to *Rubrics* only. It defines the data source used to connect to the database that stores the captions for the rubrics labels. Leave empty if you do not want to link to a database.

Delphi: `property RubricsField: string read GetDataField write SetDataField;`
C++: `__property AnsiString RubricsField = {read=GetDataField, write=SetDataField};`

This property applies to scaling 144 set to *Rubrics* only. It holds the name of the database field used to retrieve the captions for the rubrics labels. Leave empty if you do not want to link to a database.

Delphi: `property AutoTickParts: Boolean read FAutoTickParts write SetAutoTickParts;`
C++: `__property bool AutoTickParts = {read=FAutoTickParts, write=SetAutoTickParts, nodefault};`

If set to *true*, the axis scale partitions will be determined automatically depending on the values range and the size of the axis. Alternatively, you can set the partitions yourself, although this is not recommended.

Delphi: `property LabelDistanceFactor: Single read FLabelDistanceFactor write SetLabelDistanceFactor;`
C++: `__property float LabelDistanceFactor = {read=FLabelDistanceFactor, write=SetLabelDistanceFactor, nodefault};`

This factor controls the density of axes labels and following major and minor ticks at the axis. Values smaller 1 will give very dense auto tick partitions. Values higher than 1 will give less dense scaling ticks.

Delphi: `property LabelPart: Double read FLabelPart write SetLabelPart;`
C++: `__property double LabelPart = {read=FLabelPart, write=SetLabelPart};`

The setting of this property applies only if AutoTickParts 146 is set to *false*. This partition defines the positions of the major ticks showing number labels. It defines the fractions used for ticks of the normalized range covered by the current values. This means that regardless of the actual start and stop values, the axis is treated as if it were displaying values in the range 0...10, and tick positions are calculated at the position matching the values as dividers.

Delphi: `property MajorTickPart: Double read FMajorTickPart write SetMajorPart;`
C++: `__property double MajorTickPart = {read=FMajorTickPart, write=SetMajorPart};`

The setting of this property applies only if AutoTickParts 146 is set to *false*. This partition defines the positions of the major ticks.

Delphi: `property TickPart: Double read FTickPart write SetPart;`
C++: `__property double TickPart = {read=FTickPart, write=SetPart};`

The setting of this property applies only if AutoTickParts 146 is set to *false*. This partition defines the positions of the minor ticks.

Delphi:
```
TRtMajorGridMode = (AtLabel,AtMajorTick);
property MajorGridMode: TRtMajorGridMode read FMajorGridMode
write SetMajorGridMode;
```
C++:
```
enum TRtMajorGridMode { AtLabel, AtMajorTick };
__property TRtMajorGridMode MajorGridMode =
{read=FMajorGridMode, write=SetMajorGridMode, nodefault};
```

This property defines whether the major grid lines of the graph refer to the label positions or the major tick marks of the axis.

Delphi: `property DistanceValuesAxis: Single read FDistanceValuesAxis write SetDistanceValuesAxis;`
C++: `__property float DistanceValuesAxis = {read=FDistanceValuesAxis, write=SetDistanceValuesAxis};`

The distance between the axis and the series display area of the graph.

Delphi: `property DistanceAxisLabels: Single read FDistanceAxisLabels write SetDistanceAxisLabels;`
C++: `__property float DistanceAxisLabels = {read=FDistanceAxisLabels, write=SetDistanceAxisLabels};`

The distance between the axis line and ticks to the scaling number labels.

Delphi: `property DistanceAxisExtraExponent: Single read FDistanceAxisExtraExponent write SetDistanceAxisExtraExponent;`
C++: `__property float DistanceAxisExtraExponent = {read=FDistanceAxisExtraExponent, write=SetDistanceAxisExtraExponent};`

The offset added to the DistanceAxisLabels 147 for the additional exponent. The additional exponent will only be displayed if the range matches the MinForExtraExponent 144 or MaxForExtraExponent 144.

Delphi: `property DistanceNumbersCaption: Single read GetDistanceNumbersCaption write SetDistanceNumbersCaption;`
C++: `__property float DistanceNumbersCaption = {read=GetDistanceNumbersCaption, write=SetDistanceNumbersCaption};`

The distance between the scaling numbers and the caption 140 of the axis.

Delphi: `property DistanceToBorder: Single read GetDistanceToBorder write SetDistanceToBorder;`
C++: `__property float DistanceToBorder = {read=GetDistanceToBorder, write=SetDistanceToBorder};`

The distance between the caption and the border of the control. You may want to use the TRtGraph2D.Distance... 157 instead.

Delphi: `property IsSlaveAxis: Boolean read FIsSlaveAxis write SetIsSlaveAxis;`
C++: `__property bool IsSlaveAxis = {read=FIsSlaveAxis, write=SetIsSlaveAxis, nodefault};`

If set to *true*, the axis scale of a secondary axis will be synchronized with the scaling of the MasterAxis 147.

Delphi: `property MasterAxis: TRtAxis read FMasterAxis write SetMasterAxis;`
C++: `__property TRtAxis* MasterAxis = {read=FMasterAxis, write=SetMasterAxis};`

Specifies the axis control to synchronize the scaling of a secondary axis, if IsSlaveAxis 147 is set to true.

Delphi: `property SlaveOffset: Double read FSlaveOffset write SetSlaveOffset;`
C++: `__property double SlaveOffset = {read=FSlaveOffset, write=SetSlaveOffset};`

If the axis is a slave axis, the actual start and stop values can be calculated using the start and stop of the master axis:
*start = MasterAxis.Start + SlaveOffset + MasterAxis.Start*SlaveSlope*
*stop = MasterAxis.Stop + SlaveOffset + MasterAxis.Stop*SlaveSlope*
Set EndRounding 143 to *RndNothing* if you use settings other than 0.

Delphi: `property SlaveSlope: Double read FSlaveSlope write SetSlaveSlope;`
C++: `__property double SlaveSlope = {read=FSlaveSlope, write=SetSlaveSlope};`

If the axis is a slave axis, the actual start and stop values can be calculated using the start and stop of the master axis:
*start = MasterAxis.Start + SlaveOffset + MasterAxis.Start*SlaveSlope*
*stop = MasterAxis.Stop + SlaveOffset + MasterAxis.Stop*SlaveSlope*
Set EndRounding 143 to *RndNothing* if you use settings other than 1.

Delphi: `property IsClipped: Boolean read GetIsClipped;`
C++: `__property bool IsClipped = {read=GetIsClipped};`

This function determines whether the axis shows clipped values, meaning that the values range is cut by the actual start and stop values. Used internally to display optional scrollbars for graphs.

Delphi: `property Horizontal: Boolean read GetHorizontal;`
C++: `__property bool Horizontal = {read=GetHorizontal};`

This function determines whether the axis is drawn horizontally. It will return *true* with axis positions 142 *Bottom* and *Top*.

Delphi: `property CanZoomBack: Boolean read GetCanZoomBack;`
C++: `__property bool CanZoomBack = {read=GetCanZoomBack};`

This function determines whether the axis has been previously zoomed and can store any zoom stage for retrieval (for internal use only). Preferable to use TRtGraph2D.CanZoomBack 159 if needed.

Delphi: `property ZoomStage: Integer read GetCurrentZoomStage`
C++: `__property int ZoomStage = {read=GetCurrentZoomStage};`

The current index of the zoom depth of the graph. 0 means full scale. Preferable to use TRtGraph2D.ZoomStage 159 if needed.

Delphi: `property ZoomBufferSize: Integer read GetZoomBufferSize;`
C++: `__property int ZoomBufferSize = {read=GetZoomBufferSize};`

The current size of the zoom buffer. The count of zoom operations the user has executed to zoom into the graph. Preferable to use TRtGraph2D.ZoomBufferSize 159 if needed.

Events

Delphi: `property OnGetTicks: TNotifyEvent read mGetTicks write mGetTicks;`
C++: `__property Classes::TNotifyEvent OnGetTicks = {read=mGetTicks, write=mGetTicks};`

This event is always triggered before the axis tick positions are calculated. It can be used to modify partitions.

Delphi:
```
TRtScaleLabelEventArgs = record
  Value: Double;
  Caption: TRtRichCaption;
end;
TRtScaleLabelEventHandler = procedure(Sender: TObject; var e: TRtScaleLabelEventArgs) of object;
property OnGetScaleLabel: TRtScaleLabelEventHandler read mGetScaleLabel write mGetScaleLabel;
```

C++:
```
struct TRtScaleLabelEventArgs{
public:
      double Value;
      WideString Caption;
};
typedef void __fastcall (__closure
*TRtScaleLabelEventHandler)(System::TObject* Sender,
TRtScaleLabelEventArgs &e);
__property TRtScaleLabelEventHandler OnGetScaleLabel =
{read=mGetScaleLabel, write=mGetScaleLabel};
```

This event is triggered each time a scale label is required. It passes the actual value and default caption for the scale label. You can modify the caption to your own needs.

Delphi:
```
property OnCalculated: TNotifyEvent read mCalculated write
mCalculated;
```
C++:
```
__property Classes::TNotifyEvent OnCalculated =
{read=mCalculated, write=mCalculated};
```

This event is triggered each time the internal positions of the axis are calculated. You can use this to adjust the positions of other objects, for example.

Public Methods

Delphi:
```
function NumberToWorld(n: Double): Single;
```
C++:
```
float __fastcall NumberToWorld(double n);
```

This function returns the graphics world position (as Pixel) coefficient related to the axis of a number supplied as a parameter.

Delphi:
```
function WorldToNumber(n: Single): Double;
```
C++:
```
double __fastcall WorldToNumber(float n);
```

This function returns the number corresponding to a graphics world position (Pixel) coefficient related to the axis supplied as a parameter.

Delphi:
```
procedure ZoomClearBuffer;
```
C++:
```
void __fastcall ZoomClearBuffer(void);
```

Clears the internal zoom buffer (for internal use only). Preferable to use TRtGraph2D.ZoomReset 161 if needed.

Delphi:
```
procedure ZoomIn(NewStart,NewStop: Double);
```
C++:
```
void __fastcall ZoomIn(double NewStart, double NewStop);
```

Sets the new range for zoom purposes (for internal use only). Preferable to use TRtGraph2D.ZoomTo 161 if needed.

Delphi:
```
procedure ZoomBack;
```
C++:
```
void __fastcall ZoomBack(void);
```

Zooms back one stage (for internal use only). Preferable to use TRtGraph2D.ZoomBack 161 if needed.

Delphi:
```
procedure RedoZoom;
```
C++:
```
void __fastcall RedoZoom(void);
```

This method will restore the last deeper display ranges when zoomed back (Inverse of ZoomBack 149, for internal use only). Preferable to use TRtGraph2D.RedoZoom 161 if needed.

Delphi: `procedure ZoomToStage(Stage: Integer);`
C++: `void __fastcall ZoomToStage(int Stage);`

This method will restore the display ranges indicated as index from the zoom buffer when zoomed. Preferable to use TRtGraph2D.ZoomToStage 161 if needed.

Delphi: `procedure Modified;`
C++: `void __fastcall Modified(void);`

Marks the axis as needing to be recalculated and redrawn.

Delphi: `procedure RangeModified;`
C++: `void __fastcall RangeModified(void);`

Marks the axis values range as changed and the axis as needing to be recalculated and redrawn.

Delphi: `procedure SetStartStop(AStart,AStop: Double);`
C++: `void __fastcall SetStartStop(double AStart, double AStop);`

Sets the axis range directly, without updating the zoom buffer (for internal use only).

Delphi:
```
procedure WriteTo(FileName: string); overload;
procedure WriteTo(Stream: TStream); overload;
procedure WriteTo(Node: IXMLNode); overload;
procedure WriteTo(Ini: TCustomIniFile; Section: string);
overload;
```
C++:
```
void __fastcall WriteTo(AnsiString FileName);
void __fastcall WriteTo(Classes::TStream* Stream);
void __fastcall WriteTo(Xmlintf::_di_IXMLNode Node);
void __fastcall WriteTo(Inifiles::TCustomIniFile* Ini,
AnsiString Section);
```

The axis settings can be saved to a file in four different ways:
to a binary file specified by name,
to any binary stream,
to an XML document node,
to a specified section of an INI-file.

Delphi:
```
procedure ReadFrom(FileName: string); overload;
procedure ReadFrom(Stream: TStream); overload;
procedure ReadFrom(Node: IXMLNode); overload;
procedure ReadFrom(Ini: TCustomIniFile; Section: string);
overload;
```
C++:
```
void __fastcall ReadFrom(AnsiString FileName);
void __fastcall ReadFrom(Classes::TStream* Stream);
void __fastcall ReadFrom(Xmlintf::_di_IXMLNode Node);
void __fastcall ReadFrom(Inifiles::TCustomIniFile* Ini,
AnsiString Section);
```

The axis settings can be read from a file in four different ways:
from a binary file specified by name,
from any binary stream,
from an XML document node,
from a specified section of an INI file.

3.5.34 TRtCustomGraph

This class provides properties and methods common to the Cartesian graph control 157 and the pie/donut 173 chart control.

Unit/Namespace: `RtGraph2D`

Declaration

Delphi: `TRtCustomGraph = class(TCustomPanel);`
C++: `class TRtCustomGraph : public Extctrls::TCustomPanel;`

Published Properties

Delphi: `property Caption: TRtRichCaption 70 read GetCaption write SetCaption;`
C++: `__property WideString Caption = {read=GetCaption, write=SetCaption};`
↳ The optional caption displayed on top of the graph.

Delphi: `property CaptionVisible: Boolean read FCaptionVisible write SetCaptionVisible;`
C++: `__property bool CaptionVisible = {read=FCaptionVisible, write=SetCaptionVisible, nodefault};`
↳ If set to *true*, then the caption will be displayed on top of the graph.

Delphi: `property CaptionAlignment: TAlignment read FCaptionAlignment write SetCaptionAlignment`
C++: `property Classes::TAlignment CaptionAlignment = {read=FCaptionAlignment, write=SetCaptionAlignment, nodefault};`
↳ The alignment of the title caption relative to the control width.

Delphi: `property CaptionColor: TRtColor 58 read GetCaptionColor write SetCaptionColor;`
C++: `__property Rtgdi::TRtColor CaptionColor = {read=GetCaptionColor, write=SetCaptionColor, nodefault};`
↳ The color of the title caption.

Delphi: `property CaptionDistanceBefore: Single read FCaptionDistanceBefore write SetCaptionDistanceBefore;`
C++: `__property float CaptionDistanceBefore = {read=FCaptionDistanceBefore, write=SetCaptionDistanceBefore};`
↳ The distance of the title caption to the top border of the control.

Delphi: `property CaptionDistanceAfter: Single read FCaptionDistanceAfter write SetCaptionDistanceAfter;`
C++: `__property float CaptionDistanceAfter = {read=FCaptionDistanceAfter, write=SetCaptionDistanceAfter};`
↳ The distance from the lower edge of the title caption to the top of the drawing area.

Delphi: `property CaptionFont: TRtFont read GetCaptionFont write SetCaptionFont stored GetStoreFont;`
C++: `__property Rtrichlabel::TRtFont* CaptionFont = {read=GetCaptionFont, write=SetCaptionFont, stored=GetStoreFont};`
↳ The font with enhanced settings used for the title caption 151.

Delphi: `property BackColor: TRtColor read FBackColor write SetBackColor;`
C++: `__property Rtgdi::TRtColor BackColor = {read=FBackColor, write=SetBackColor, nodefault};`
The background color of the graph.

Delphi: `property BackgroundGradient: TRtGradientSettings read FBackgroundGradient write FBackgroundGradient;`
C++: `__property TRtGradientSettings* BackgroundGradient = {read=FBackgroundGradient, write=FBackgroundGradient};`
Settings for an optional gradient drawn at the background of the graph.

Delphi:
```
TRtSmoothingMode = (None, HighSpeed, AntiAlias, HighQuality);
property SmoothingMode: TRtSmoothingMode read FSmoothingMode
write SetSmoothingMode;
```
C++:
```
enum TRtSmoothingMode { None, HighSpeed, AntiAlias,
HighQuality };
__property TRtSmoothingMode SmoothingMode =
{read=FSmoothingMode, write=SetSmoothingMode, nodefault};
```
Specifies the type of smoothing (antialiasing) that is applied to lines and curves and edges of areas.

> This option highly influences the performance of drawing. Setting smoothing to *HighQuality* will produce the best looking output. But if you handle large amounts of data it might be better to set to *None*. This will speed up the drawing time to less than the half. If the line style is solid or the line width is 1, the lines drawing will be nearly instantaneous.

Delphi: `property UndoStack: TRtUndoStack read FUndoStack write SetUndoStack;`
C++: `__property Rtundostack::TRtUndoStack* UndoStack = {read=FUndoStack, write=SetUndoStack};`
The undo stack object is used to store property changes automatically. Set to *nil* if you do not need any undo functionality or if you want to handle this task yourself. Setting this property will also set the undo stack used to store the properties of the axes, series legends and markers within the graph.

Public Methods

Delphi: `procedure Modified; virtual;`
C++: `virtual void __fastcall Modified(void);`
This method marks the graph as needing to be recalculated and redrawn.

Delphi:
```
procedure WriteTo(FileName: string; IncludeValues: Boolean =
False); overload;
procedure WriteTo(Stream: TStream; IncludeValues: Boolean =
False); overload; virtual;
procedure WriteTo(Node: IXMLNode; IncludeValues: Boolean =
False); overload; virtual;
procedure WriteTo(Ini: TCustomIniFile; Section: string;
IncludeValues: Boolean = False); overload; virtual;
```

C++:
```
void __fastcall WriteTo(AnsiString FileName, bool
IncludeValues = false);
virtual void __fastcall WriteTo(Classes::TStream* Stream,
bool IncludeValues = false);
virtual void __fastcall WriteTo(Xmlintf::_di_IXMLNode Node,
bool IncludeValues = false);
virtual void __fastcall WriteTo(Inifiles::TCustomIniFile*
Ini, AnsiString Section, bool IncludeValues = false);
```

The graph settings can be saved to a file in four different ways:
- to a binary file specified by name,
- to any binary stream,
- to an XML document node,
- to a specified section of an INI-file

It will store all the graph properties.

TRtGraph2D will store all the axes properties, all the series properties, all the legend properties and all the markers properties contained in the graph. If you set ***IncludeValues*** to *true*, then the data vectors used with the series will also be saved.

TRtPieChart will also store all the pie legend properties and fill palette settings. If you set ***IncludeValues*** to *true*, then the ***Values***. ***ExplodePercent*** and ***Captions*** data will also be saved.

This will allow you to save or load 153 a graph with just one line of code.

Delphi:
```
procedure ReadFrom(FileName: string); overload;
procedure ReadFrom(Stream: TStream); overload; virtual;
procedure ReadFrom(Node: IXMLNode); overload; virtual;
procedure ReadFrom(Ini: TCustomIniFile; Section: string);
overload; virtual;
```

C++:
```
void __fastcall ReadFrom(AnsiString FileName);
virtual void __fastcall ReadFrom(Classes::TStream* Stream);
virtual void __fastcall ReadFrom(Xmlintf::_di_IXMLNode Node);
virtual void __fastcall ReadFrom(Inifiles::TCustomIniFile*
Ini, AnsiString Section);
```

The graph settings can be read from a file in four different ways:
- from a binary file specified by name,
- from any binary stream,
- from an XML document node,
- from a specified section of an INI file.

It will restore all the graph properties.

TRtGraph2D will restore all the axes properties, all the series properties, all the legend properties and all the markers properties contained in the graph. If you saved 152 the file and set ***IncludeValues*** to *true*, then the data vectors used with the series will also be restored.

TRtPieChart will also restore all the palette settings all legend properties. If you saved 152 the file and set ***IncludeValues*** to *true*, then the data for ***Values***, ***ExplodePercent*** and ***Captions*** will also be restored.

Delphi:
```
function AsTMetafile: TMetafile;
```

C++:
```
Graphics::TMetafile* __fastcall AsTMetafile(void);
```

This function creates a new instance of a **TMetafile** containing the graph as image. If you need to store the metafile, it is recommended that you use SaveImage 154 instead, because it can yield higher-quality output.

Delphi: `function AsTBitmap(TransparentExchangeColor: TColor = clWhite): TBitmap;`
C++: `Graphics::TBitmap* __fastcall AsTBitmap(Graphics::TColor TransparentExchangeColor = (Graphics::TColor)(0xffffff));`

This function creates a new instance of a **TBitmap** containing the graph as an image. If the graph uses a transparent background, this will be exchanged with the ***TransparentExchangeColor***. If you need to store the bitmap, it is recommended to use SaveImage 154 instead, because it can yield higher-quality output.

Delphi: `procedure CopyToClipboard; virtual;`
C++: `virtual void __fastcall CopyToClipboard(void);`

This method will copy the graph image to the Windows clipboard using the enhanced metafile format.

Delphi:
```
TRtImageFileFormat = (EMF,BMP,JPEG,PNG,GIF,TIFF,EXIF,Icon);
procedure SaveImage(Stream: TStream; Format:
TRtImageFileFormat; RemoveTransparency: Boolean = False;
TransparentExchangeColor: TColor = clWhite); overload;
virtual;
procedure SaveImage(Filename: string; Format:
TRtImageFileFormat; RemoveTransparency: Boolean = False;
TransparentExchangeColor: TColor = clWhite); overload;
```
C++:
```
enum TRtImageFileFormat { EMF, BMP, JPEG, PNG, GIF, TIFF,
EXIF, Icon };
virtual void __fastcall SaveImage(Classes::TStream* Stream,
TRtImageFileFormat Format, bool RemoveTransparency = false,
Graphics::TColor TransparentExchangeColor = (Graphics::
TColor)(0xffffff));
void __fastcall SaveImage(AnsiString Filename,
TRtImageFileFormat Format, bool RemoveTransparency = false,
Graphics::TColor TransparentExchangeColor = (Graphics::
TColor)(0xffffff));
```

This method will store an image of the graph either to a file specified by name or to an arbitrary stream. You can specify the output format as one of EMF, BMP...Icon. By default, the image is stored using transparency for the bitmap formats. If you define ***RemoveTransparency*** as *true*, the method will use the color defined as ***TransparentExchangeColor*** instead of a transparent background.

Delphi: `procedure PrintTo(APrinter: TPrinter; Rect: TRect); virtual;`
C++: `virtual void __fastcall PrintTo(Printers::TPrinter* APrinter, const Types::TRect &Rect);`

This method will print the graph to a printer canvas. The output will fill the rectangle defined as a parameter. The graph will retain the aspect when printing. This means that although the aspect of the print output rectangle can differ to the graph aspect, the printout will be optimized to fit the width/height of the rectangle, while retaining the graph aspect. See the "Printing" example to see how this method can be applied.

Delphi: `function Updating: Boolean; reintroduce;`
C++: `HIDESBASE bool __fastcall Updating(void);`

This function determines whether the automatic update mechanism is currently disabled.

Delphi: `procedure BeginUpdate;`
C++: `void __fastcall BeginUpdate(void);`

This method can be used to temporarily disable the internal automatic updating mechanisms. Call EndUpdate 155 afterwards to re-enable them. Calls to BeginUpdate/EndUpdate can be nested.

Delphi: `procedure EndUpdate;`
C++: `void __fastcall EndUpdate(void);`

↳ This method can be used to re-enable the internal automatic updating mechanisms after BeginUpdate 154 has been called. afterwards to re-enable them. Calls to BeginUpdate/EndUpdate can be nested.

Delphi: `procedure Calculate; virtual;`
C++: `virtual void __fastcall Calculate(void);`

↳ This method performs all necessary calculations for the graph (for internal use only).

3.5.35 TRtZoomSettings Class

This class is used to store the zooming options for the graph in a more organized way.

Unit/Namespace: `RtGraph2D`

Declaration

Delphi: `TRtZoomSettings = class(TPersistent)`
C++: `class TRtZoomSettings : public Classes::TPersistent;`

Published Properties

Delphi: `property Auto: Boolean read FAuto write FAuto;`
C++: `__property bool Auto = {read=FAuto, write=FAuto, nodefault};`

↳ If set to *true*, then auto zooming and panning is enabled. You can drag a zoom frame and pan zoomed graph displays with the mouse.

Delphi: `property OptimizeSeries: Boolean read FOptimizeSeries write FOptimizeSeries;`
C++: `__property bool OptimizeSeries = {read=FOptimizeSeries, write=FOptimizeSeries, nodefault};`

↳ If set to *true*, then the zoomed range will be optimized for the series data points visible inside the dragged zoom frame.

Delphi: `TRtZoomBufferType = (AsCircularBuffer, AsStack);`
`property BufferType: TRtZoomBufferType read FBufferType write FBufferType;`
C++: `__property TRtZoomBufferType BufferType = {read=FBufferType, write=FBufferType, nodefault};`

↳ Defines whether the zoom range data are stored to a circular buffer or to a stack. If the data are stored *AsStack* the user cannot ***ZoomBack*** on stage 0 and cannot ***RedoZoom*** on the last stage. If the data are stored *AsCircularBuffer* the user will go to the last stage or first stage respectively.

Delphi: `property BackOnClickBeside: Boolean read FBackOnClickBeside write FBackOnClickBeside;`
C++: `__property bool BackOnClickBeside = {read=FBackOnClickBeside, write=FBackOnClickBeside, nodefault};`

↳ If set to *true* and the graph has already been zoomed, you can ZoomBack 161 by clicking on any space outside the graph series drawing area.

Delphi: `property BackOnRightBottomLeftTopFrame: Boolean read FBackOnRightBottomLeftTopFrame write FBackOnRightBottomLeftTopFrame;`
C++: `__property bool BackOnRightBottomLeftTopFrame = {read=FBackOnRightBottomLeftTopFrame, write=FBackOnRightBottomLeftTopFrame, nodefault};`

If set to *true* and the graph has already been zoomed, you can ZoomBack 161 by dragging a frame from bottom right to top left inside the graph series drawing area. Remember that then zooming in uses the opposite direction of dragging the frame.

Delphi: `property ResetOnDoubleClickBeside: Boolean read FResetOnDoubleClickBeside write FResetOnDoubleClickBeside;`
C++: `__property bool ResetOnDoubleClickBeside = {read=FResetOnDoubleClickBeside, write=FResetOnDoubleClickBeside, nodefault};`

If set to *true* and the graph has already been zoomed, you can ZoomReset 161 by double-clicking on any space outside the graph series drawing area.

Delphi: `property ResetOnRightBottomLeftTopFrame: Boolean read FResetOnRightBottomLeftTopFrame write FResetOnRightBottomLeftTopFrame;`
C++: `__property bool ResetOnRightBottomLeftTopFrame = {read=FResetOnRightBottomLeftTopFrame, write=FResetOnRightBottomLeftTopFrame, nodefault};`

If set to *true* and the graph has already been zoomed, you can ZoomReset 161 by dragging a frame from bottom right to the top left outside the graph series drawing area.

Delphi: `property StartZoom: TShiftState read FZoomShift write SetZoomShift;`
C++: `__property Classes::TShiftState StartZoom = {read=FZoomShift, write=SetZoomShift, nodefault};`

The mouse button control key combination that starts the zoom drag frame operation.

Delphi: `property StartPanning: TShiftState read FPanningShift write SetPanningShift;`
C++: `__property Classes::TShiftState StartPanning = {read=FPanningShift, write=SetPanningShift, nodefault};`

The mouse button control key combination that starts the panning operation if the graph is already zoomed.

Delphi: `property PanningCursor: TCursor read FPanningCursor write FPanningCursor;`
C++: `__property Controls::TCursor PanningCursor = {read=FPanningCursor, write=FPanningCursor, nodefault};`

The mouse cursor shown while panning the zoomed graph.

Delphi: `property HScrollbarVisible: Boolean read FHScrollbarVisible write FHScrollbarVisible;`
C++: `__property bool HScrollbarVisible = {read=FHScrollbarVisible, write=FHScrollbarVisible, nodefault};`

If set to *true*, then the graph will show a horizontal scrollbar if the X-range was zoomed.

Delphi: `property VScrollbarVisible: Boolean read FVScrollbarVisible write FVScrollbarVisible;`
C++: `__property bool VScrollbarVisible = {read=FVScrollbarVisible, write=FVScrollbarVisible, nodefault};`

If set to *true*, then the graph will show a vertical scrollbar if the Y-range was zoomed.

Delphi: `TRtScrollIncrement = (Page, MajorTick, MinorTick, Pixel);`
`property ScrollSmallIncrement: TRtScrollIncrement read FScrollSmallIncrement write FScrollSmallIncrement;`
C++: `enum TRtScrollIncrement { Page, MajorTick, MinorTick, Pixel };`
`__property TRtScrollIncrement ScrollSmallIncrement = {read=FScrollSmallIncrement, write=FScrollSmallIncrement, nodefault};`

The amount the graph is scrolled by clicking on the arrow buttons of the scrollbars.

Delphi: `property ScrollLargeIncrement: TRtScrollIncrement read FScrollLargeIncrement write FScrollLargeIncrement;`
C++: `__property TRtScrollIncrement ScrollLargeIncrement = {read=FScrollLargeIncrement, write=FScrollLargeIncrement, nodefault};`

The amount the graph is scrolled by clicking next to the scrollbar sliders.

3.5.36 TRtGraph2D Class

This class provides a component that acts as a container for all the other 2D components such as axes, legends and movable markers. It also provides functionality for drawing the assigned series inside the graph drawing area.

Unit/Namespace: `RtGraph2D`

Declaration

Delphi: `TRtGraph2D = class(TRtCustomGraph);` 151
C++: `class TRtGraph2D : public Rtgraph2d::TRtCustomGraph;`

Public Properties

Delphi: `property HorizontalScrollBar: TScrollBar read FHScrollBar;`
C++: `__property Stdctrls::TScrollBar* HorizontalScrollBar = {read=FHScrollBar};`

The horizontal scrollbar visible when zoomed (for internal use only).

Delphi: `property VerticalScrollbar: TScrollBar read FVScrollBar;`
C++: `__property Stdctrls::TScrollBar* VerticalScrollbar = {read=FVScrollbar};`

The vertical scrollbar visible when zoomed (for internal use only).

Delphi: `property DataAreaColor: TRtColor read FDataAreaColor write SetDataAreaColor;` 58
C++: `__property Rtgdi::TRtColor DataAreaColor = {read=FDataAreaColor, write=SetDataAreaColor, nodefault};`

The background color of the series drawing area.

Delphi: `property DataAreaGradient: TRtGradientSettings read FDataAreaGradient write FDataAreaGradient;`
C++: `__property TRtGradientSettings* DataAreaGradient = {read=FDataAreaGradient, write=FDataAreaGradient};`
Settings for an optional gradient drawn at the background of the graph.

Delphi: `property DataAreaFrame: TRtLineSettings read FDataAreaFrame write SetDataAreaFrame;`
C++: `__property Rtstyles::TRtLineSettings* DataAreaFrame = {read=FDataAreaFrame, write=SetDataAreaFrame};`
The settings used to draw the border line of the series drawing area.

Delphi: `property DistanceBorderAxisTop: Single read FDistanceBorderAxisTop write SetDistanceBorderAxisTop;`
C++: `__property float DistanceBorderAxisTop = {read=FDistanceBorderAxisTop, write=SetDistanceBorderAxisTop};`
The distance of the graph border to the top axis or the series drawing area.

Delphi: `property DistanceBorderAxisBottom: Single read FDistanceBorderAxisBottom write SetDistanceBorderAxisBottom;`
C++: `__property float DistanceBorderAxisBottom = {read=FDistanceBorderAxisBottom, write=SetDistanceBorderAxisBottom};`
The distance of the graph border to the bottom axis or the series drawing area.

Delphi: `property DistanceBorderAxisLeft: Single read FDistanceBorderAxisLeft write SetDistanceBorderAxisLeft;`
C++: `__property float DistanceBorderAxisLeft = {read=FDistanceBorderAxisLeft, write=SetDistanceBorderAxisLeft};`
The distance of the graph border to the left axis or the series drawing area.

Delphi: `property DistanceBorderAxisRight: Single read FDistanceBorderAxisRight write SetDistanceBorderAxisRight;`
C++: `__property float DistanceBorderAxisRight = {read=FDistanceBorderAxisRight, write=SetDistanceBorderAxisRight};`
The distance of the graph border to the right axis or the series drawing area.

Delphi: `property PrimaryXAxis: TRtAxis read FPrimaryXAxis write SetPrimaryXAxis;`
C++: `__property Rtaxis::TRtAxis* PrimaryXAxis = {read=FPrimaryXAxis, write=SetPrimaryXAxis};`
The axis used as the primary x-axis, giving the positions for the vertical grid lines.

Delphi: `property PrimaryYAxis: TRtAxis read FPrimaryYAxis write SetPrimaryYAxis;`
C++: `__property Rtaxis::TRtAxis* PrimaryYAxis = {read=FPrimaryYAxis, write=SetPrimaryYAxis};`
The axis used as the primary y-axis, giving the positions for the horizontal grid lines.

Delphi: `property MajorGridHorizontal: TRtLineSettings read FMajorGridHorizontal write FMajorGridHorizontal;`
C++: `__property Rtstyles::TRtLineSettings* MajorGridHorizontal = {read=FMajorGridHorizontal, write=FMajorGridHorizontal};`

- The settings of the optional horizontal grid lines at the positions of the labels or major ticks of the primary y-axis.

Delphi: `property MinorGridHorizontal: TRtLineSettings read FMinorGridHorizontal write FMinorGridHorizontal;`
C++: `__property Rtstyles::TRtLineSettings* MinorGridHorizontal = {read=FMinorGridHorizontal, write=FMinorGridHorizontal};`

- The settings of the optional horizontal grid lines at the positions of the minor ticks of the primary y-axis.

Delphi: `property MajorGridVertical: TRtLineSettings read FMajorGridVertical write FMajorGridVertical;`
C++: `__property Rtstyles::TRtLineSettings* MajorGridVertical = {read=FMajorGridVertical, write=FMajorGridVertical};`

- The settings of the optional vertical grid lines at the positions of the labels or major ticks of the primary x-axis.

Delphi: `property MinorGridVertical: TRtLineSettings read FMinorGridVertical write FMinorGridVertical;`
C++: `__property Rtstyles::TRtLineSettings* MinorGridVertical = {read=FMinorGridVertical, write=FMinorGridVertical};`

- The settings of the optional vertical grid lines at the positions of the minor ticks of the primary x-axis.

Delphi: `property ZeroOrigin: Boolean read FZeroOrigin write SetZeroOrigin;`
C++: `__property bool ZeroOrigin = {read=FZeroOrigin, write=SetZeroOrigin, nodefault};`

- If set to *true*, then the primary x- and y-axis will be arranged to cross at (0, 0).

Delphi: `property Zoom: TRtZoomSettings read FZoom write FZoom;`
C++: `__property TRtZoomSettings* Zoom = {read=FZoom, write=FZoom};`

- The auto zoom options.

Delphi: `property CanZoomBack: Boolean read GetCanZoomBack;`
C++: `__property bool CanZoomBack = {read=GetCanZoomBack};`

- This function determines whether the graph has been zoomed and whether the last ranges can be restored.

Delphi: `function CanRedoZoom: Boolean;`
C++: `__property bool CanRedoZoom = {read=GetCanRedoZoom};`

- This function determines whether the graph has been zoomed back and whether the last deeper ranges can be restored (Inverse of CanZoomBack).

Delphi: `property ZoomBufferSize: Integer read GetZoomBufferSize;`
C++: `__property int ZoomBufferSize = {read=GetZoomBufferSize};`

- The current size of the zoom buffer. The count of zoom operations the user has executed to zoom into the graph.

Delphi: `property ZoomStage: Integer read GetCurrentZoomStage`
C++: `__property int ZoomStage = {read=GetCurrentZoomStage};`

- The current index of the zoom depth of the graph. 0 means full scale.

Events

Delphi: `property OnBeginZoom: TNotifyEvent read mBeginZoom write mBeginZoom;`

C++: `__property Classes::TNotifyEvent OnBeginZoom = {read=mBeginZoom, write=mBeginZoom};`

This event is triggered when the user starts 156 to drag the zoom frame. It can be used to store undo information, for example.

Delphi: `property OnEndZoom: TNotifyEvent read mEndZoom write mEndZoom;`

C++: `__property Classes::TNotifyEvent OnEndZoom = {read=mEndZoom, write=mEndZoom};`

This event is triggered when the user stops dragging the zoom frame.

Delphi: `property OnBeginPanning: TNotifyEvent read mBeginPanning write mBeginPanning;`

C++: `__property Classes::TNotifyEvent OnBeginPanning = {read=mBeginPanning, write=mBeginPanning};`

This event is triggered when the user starts 156 to pan the zoomed graph. It can be used to store undo information, for example.

Delphi: `property OnEndPanning: TNotifyEvent read mEndPanning write mEndPanning;`

C++: `__property Classes::TNotifyEvent OnEndPanning = {read=mEndPanning, write=mEndPanning};`

This event is triggered when the user stops panning the zoomed graph.

Public Methods

Delphi:
```
TRtFindSeriesPointResult = record
  Found: Boolean;
  InSeries: TObject;
  Index: Integer;
  X,Y: Double;
end;
function FindSeriesPoint(MouseX,MouseY: Integer):
TRtFindSeriesPointResult;
```

C++:
```
struct TRtFindSeriesPointResult{
public:
      bool Found;
      System::TObject* InSeries;
      int Index;
      double X;
      double Y;
};
TRtFindSeriesPointResult __fastcall FindSeriesPoint(int
MouseX, int MouseY);
```

This function can be used to find the series and the data point indicated by a mouse position passed as X and Y coordinates. The function determines whether there is a data point under the mouse, the series containing the data point, the index to the point and the position as a data point scaled to X and Y. It can be used to implement a status display on mouse movement (see the "FinancialSeries" example in the "Financial" directory).

Delphi: `procedure SeriesModified;`
C++: `void __fastcall SeriesModified(void);`
This method marks the graph as having added or removed series and as needing to be recalculated and redrawn (for internal use only).

Delphi: `procedure LegendsModified;`
C++: `void __fastcall LegendsModified(void);`
This method marks the graph as having added or removed legends and as needing to be recalculated and redrawn.

Delphi: `procedure SortLayers;`
C++: `void __fastcall SortLayers(void);`
This method is called to refresh the order of the series to display, if the layering has changed (for internal use only).

Delphi: `procedure UndoPushSeriesVectors;`
C++: `void __fastcall UndoPushSeriesVectors(void);`
This method stores the state of all data vectors assigned to the graph series onto the undo stack. If you make any changes to the data afterwards, such as adding, recalculating or deleting values, you can restore the state before the changes by calling the ***Undo*** method of the component referenced by the ***UndoStack*** property.

Delphi: `procedure ZoomTo(Corner1,Corner2: TPoint);`
C++: `void __fastcall ZoomTo(const Types::TPoint &Corner1, const Types::TPoint &Corner2)`
This methods zooms the graph to the frame defined by the corner points. It is used internally for the auto zooming.

Delphi: `procedure ZoomBack;`
C++: `void __fastcall ZoomBack(void);`
This method will restore the last display ranges when zoomed.

Delphi: `procedure RedoZoom;`
C++: `void __fastcall RedoZoom(void);`
This method will restore the last deeper display ranges when zoomed back (Inverse of ZoomBack161).

Delphi: `procedure ZoomToStage(Stage: Integer);`
C++: `void __fastcall ZoomToStage(int Stage);`
This method will restore the display ranges indicated as index from the zoom buffer when zoomed.

Delphi: `procedure ZoomReset;`
C++: `void __fastcall ZoomReset(void);`
This method resets the internal zoom buffers and will rescale the graph to once more display all series data points.

3.5.37 TRtGraphSettingsTool Class

This class provides a tool window for setting-up all the properties of the graph, the axes, the series and the legend.

Unit/Namespace: `RtGraphSettingsTool`

Declaration

Delphi: `TRtGraphSettingsTool = class(TComponent);`
C++: `class TRtGraphSettingsTool : public Classes::TComponent;`

Published Properties

Delphi: `property Caption: string read FCaption write SetCaption;`
C++: `__property AnsiString Caption = {read=FCaption, write=SetCaption};`

The caption used in the title of the tool window.

Delphi: `property Graph: TRtGraph2D read FGraph write SetGraph;`
C++: `__property Rtgraph2d::TRtGraph2D* Graph = {read=FGraph, write=SetGraph};`

The graph containing the properties to be set.

Delphi: `property ExtendedAxisOptions: Boolean read FExtendedAxisOptions write SetExtendedAxisOptions;`
C++: `__property bool ExtendedAxisOptions = {read=FExtendedAxisOptions, write=SetExtendedAxisOptions, nodefault};`

If set to *true*, then the tool window will allow the user to set up additional "enhanced" options, such as separate auto end settings for start and stop, end rounding selection, slave axis settings and moving slit options.

Delphi: `property ExtendedBarLabelOptions: Boolean read FExtendedBarLabelOptions write SetExtendedBarLabelOptions;`
C++: `__property bool ExtendedBarLabelOptions = {read=FExtendedBarLabelOptions, write=SetExtendedBarLabelOptions, nodefault};`

If set to *true*, then the tool window will allow the user to set up additional "expert" options, such as selecting whether the labels are set by the values or the captions list and also the number format for the labels.

Delphi: `property ZoomScrollPanningOptions: Boolean read FZoomScrollPanningOptions write SetZoomScrollPanningOptions;`
C++: `__property bool ZoomScrollPanningOptions = {read=FZoomScrollPanningOptions, write=SetZoomScrollPanningOptions, nodefault};`

If set to *true*, then the tool window will allow the user to set up the mouse and scroll options for auto zooming and panning.

Delphi: `property bool ShowOutliersSupport = {read=FShowOutliersSupport, write=SetShowOutliersSupport, nodefault};`
C++: `__property bool ShowOutliersSupport = {read=F ShowOutliersSupport, write=SetShowOutliersSupport, nodefault};`

If set to *true*, then the tool window will allow the user to set up whether outliers are displayed or hidden in series options.

Delphi: `property InteractiveCalculationRanges: Boolean read FInteractiveCalculationRanges write SetInteractiveCalculationRanges;`
C++: `__property bool InteractiveCalculationRanges = {read=FInteractiveCalculationRanges, write=SetInteractiveCalculationRanges, nodefault};`

- If set to *true*, then the tool window will allow the user to set up calculation and display ranges interactively for function_series 119. Two vertical markers can be dragged around the graph to define new ranges. If a range has been changed the corresponding AutoStart 120/AutoStop 120 or DisplayAutoStart 120/DisplayAutoStop 121 properties will be set to *false*. If this option is set to *false*, check boxes give access to the Auto... properties.

Delphi: `property ShowFitCovariance: Boolean read FShowFitCovariance write SetShowFitCovariance;`
C++: `__property bool ShowFitCovariance = {read=FShowFitCovariance, write=SetShowFitCovariance, nodefault};`

- If set to *true*, then the tool window will allow the user to set up DoCovariance 126 on TRtFittedLine 125 series.

Public Method

Delphi: `procedure Show;`
C++: `void __fastcall Show(void);`

- Call this method to display the settings tool window.

3.5.38 TRtMovable Class

This class is the ancestor of all movable_marker 37 components. It provides all common properties and methods needed to handle movable objects within a graph.

Unit/Namespace: `RtMarkers`

Declaration

Delphi: `TRtMovable = class(TGraphicControl);`
C++: `class TRtMovable : public Controls::TGraphicControl;`

Protected Properties

Delphi: `property XAxis: TRtAxis 138 read FXAxis write SetXAxis;`
C++: `__property Rtaxis::TRtAxis* XAxis = {read=FXAxis, write=SetXAxis};`

- The axis used to scale the X 163 parameter for the position of the hot point of the marker.

Delphi: `property YAxis: TRtAxis 138 read FYAxis write SetYAxis;`
C++: `__property Rtaxis::TRtAxis* YAxis = {read=FYAxis, write=SetYAxis};`

- The axis used to scale the Y 164 parameter for the position of the hot point of the marker.

Delphi: `property X: Double read FX write SetX;`
C++: `__property double X = {read=FX, write=SetX};`

- The X parameter using the XAxis 163 scaling for the position of the hot point of the marker.

Delphi: `property Y: Double read FY write SetY;`
C++: `__property double Y = {read=FY, write=SetY};`
↳ The Y parameter using the YAxis 163 scaling for the position of the hot point of the marker.

Delphi: `property Line: TRtSimpleLineSettings 99 read FLineSettings write FLineSettings;`
C++: `__property Rtstyles::TRtSimpleLineSettings* Line = {read=FLineSettings, write=FLineSettings};`
↳ The line style used to draw the marker.

Delphi: `property CaptionPosition: TRtContentAlignment 71 read FCaptionPosition write SetCaptionPosition;`
C++: `__property Rtrichlabel::TRtContentAlignment CaptionPosition = {read=FCaptionPosition, write=SetCaptionPosition, nodefault};`
↳ The position of the caption relative to the line end/mid.

Published Properties

Delphi: `property Caption: TRtRichCaption 70 read GetCaption write SetCaption;`
C++: `__property WideString Caption = {read=GetCaption, write=SetCaption};`
↳ The optional caption displayed at the CaptionPosition 164.

Delphi: `property CaptionVisible: Boolean read FCaptionVisible write SetCaptionVisible;`
C++: `__property bool CaptionVisible = {read=FCaptionVisible, write=SetCaptionVisible, nodefault};`
↳ If set to *true*, then the caption will be displayed with the line.

Delphi: `property CaptionVertical: Boolean read FCaptionVertical write SetCaptionVertical;`
C++: `__property bool CaptionVertical = {read=FCaptionVertical, write=SetCaptionVertical, nodefault};`
↳ If set to *true*, then the optional caption will be displayed vertically.

Delphi: `property CaptionDeltaX: Integer read FCaptionDeltaX write SetCaptionDeltaX;`
C++: `__property int CaptionDeltaX = {read=FCaptionDeltaX, write=SetCaptionDeltaX, nodefault};`
↳ The X-offset of the caption in relation to the CaptionPosition 164.

Delphi: `property CaptionDeltaY: Integer read FCaptionDeltaY write SetCaptionDeltaY;`
C++: `__property int CaptionDeltaY = {read=FCaptionDeltaY, write=SetCaptionDeltaY, nodefault};`
↳ The Y-offset of the caption in relation to the CaptionPosition 164.

Delphi: `property ForeColor: TRtColor 58 read FForeColor write SetForeColor;`
C++: `__property Rtgdi::TRtColor ForeColor = {read=FForeColor, write=SetForeColor, nodefault};`
↳ The foreground color of the control. It is identical to the color of the caption.

Delphi: `property Font: TRtFont read FFont write SetFont stored GetStoreFont;`
C++: `__property Rtrichlabel::TRtFont* Font = {read=FFont, write=SetFont, stored=GetStoreFont};`
↳ The font with enhanced settings used for the optional caption.

Delphi: `property CaptionBackColor: TRtColor read FCaptionBackColor write SetCaptionBackColor;`
C++: `__property Rtgdi::TRtColor CaptionBackColor = {read=FCaption BackColor, write=SetCaptionBackColor, nodefault};`
↳ The color of the background used to draw the caption if *CaptionBackGradient*. *Visible* = *false*. Captions are only drawn if *CaptionVisible* = *True.*

Delphi: `property CaptionBackGradient: TRtGradientSettings read FCaptionBackGradient write SetCaptionBackGradient;`
C++: `__property Rtstyles::TRtGradientSettings* CaptionBack = {read=FCaptionBackGradient, write=SetCaptionBackGradient};`
↳ Settings for the optional gradient to be drawn as background. Captions are only drawn if *CaptionVisible* = *True.*

Delphi: `property CaptionBackFrame: TRtLineSettings read FCaptionBackFrame write SetCaptionBackFrame;`
C++: `__property Rtstyles::TRtLineSettings* CaptionBackFrame = {read=FCaptionBackFrame, write=SetCaptionBackFrame};`
↳ Settings for the drawing of a border frame around the caption. Captions are only drawn if *CaptionVisible* = *True.*

Delphi: `property CaptionBackRaised: Boolean read FCaptionBackRaised write SetCaptionBackRaised;`
C++: `__property bool CaptionBackRaised = {read=FCaptionBackRaised, write=SetCaptionBackRaised, nodefault};`
↳ If this property is set to *true* the caption back will be drawn with a raised border. Captions are only drawn if *CaptionVisible* = *True.*

Delphi: `property CaptionBackLowered: Boolean read FCaptionBackLowered write SetCaptionBackLowered;`
C++: `__property bool CaptionBackLowered = {read=F CaptionBackLowered, write=SetCaptionBackLowered, nodefault};`
↳ If this property is set to *true* the caption back will be drawn with a lowered border. Captions are only drawn if *CaptionVisible* = *True.*

Delphi: `property CaptionBackShadow: Boolean read FCaptionBackShadow write SetCaptionBackShadow;`
C++: `__property bool CaptionBackShadow = {read=FCaptionBackShadow, write=SetCaptionBackShadow, nodefault};`
↳ If this property is set to *true* the caption will be drawn with a shadow to the background. Captions are only drawn if *CaptionVisible* = *True.*

Delphi: `property CaptionBackSpaceToBorder: Single read FCaptionBackSpaceToBorder write SetCaptionBackSpaceToBorder;`
C++: `__property float CaptionBackSpaceToBorder = {read=F CaptionBackSpaceToBorder, write=SetCaptionBackSpaceToBorder};`
↳ The distance from the text to the caption back border in percent of the label font size. Captions are only drawn if *CaptionVisible* = *True.*

Delphi: `property CaptionBackRounding: Single read FCaptionBackRounding write SetCaptionBackRounding;`
C++: `__property float CaptionBackRounding = {read=FCaptionBackRounding, write=SetCaptionBackRounding};`

The radius used to draw the caption back border round corners in percent of the label font size. Captions are only drawn if *CaptionVisible* 164 = *True.*

Delphi: `property ClipToData: Boolean read FClipToData write SetClipToData;`
C++: `__property bool ClipToData = {read=FClipToData, write=SetClipToData, nodefault};`

If set to *true*, then the display of the marker will be clipped so as to be only visible in the graph series drawing area.

Delphi: `property MouseDrag: Boolean read FMouseDrag write FMouseDrag;`
C++: `__property bool MouseDrag = {read=FMouseDrag, write=FMouseDrag, nodefault};`

If set to *true*, then the marker can be dragged around with the mouse during runtime.

Delphi: `property StartMovingKeys: TShiftState read FStartMoveKeys write FStartMoveKeys;`
C++: `__property Classes::TShiftState StartMovingKeys = {read=FStartMoveKeys, write=FStartMoveKeys, nodefault};`

The mouse button control key combination that starts the marker dragging operation.

Delphi: `property SnappedIdx: Integer read FSnappedIdx write SetSnappedIdx;`
C++: `__property int SnappedIdx = {read=FSnappedIdx, write=SetSnappedIdx, nodefault};`

If SnapToSeriesValues 166 has been set to *true*, this property will hold the index to the data point of the SnappedSeries 166 that is covered by the current marker hot spot position. This can be used to update a status display for the current position. The property returns *-1* if there is no data point in the range.
If you set the property to a value within the series data points index the marker will synchronize its position automatically with the current value of the point. If the value is deleted 97 the marker will also be deleted. If the SnappedSeries OutliersVisible 105 property is set to *false* and the corresponding data point at SnappedIdx is marked as Outlier 95 the marker will be invisible.

Delphi: `property SnapToSeriesValues: Boolean read FSnapToSeries write FSnapToSeries;`
C++: `__property bool SnapToSeriesValues = {read=FSnapToSeries, write=FSnapToSeries, nodefault};`

If set to *true*, then the hot spot position of the marker will snap to the series data points visible in the graph.

Delphi: `property SnappedSeries: TRtSeries read GetSnappedSeries write SetSnappedSeries;`
C++: `__property Rtseries::TRtSeries* SnappedSeries = {read=GetSnappedSeries, write=SetSnappedSeries};`

This property has a two-fold functionality. If you set it to *nil*, the marker position can snap to any of the overlaid series visible in the graph. Reading the property will return the series currently in the focus of the marker. If you set it to a series, the marker will snap only to the series data points specified.

Delphi: `property SnapToFunctions: Boolean read FSnapToFunctions write SetSnapToFunctions;`

C++: `__property bool SnapToFunctions = {read=FSnapToFunctions, write=SetSnapToFunctions, nodefault};`

- If set to *true*, then the hot spot position of the marker will snap to the series data points of calculated lines 119 e.g. interpolation values. This will result in the effect that the marker will follow the function line while dragging the marker with the mouse. If you only want to snap to the source data points you can set ***SnapToFunctions*** to *False* and ***SnappedSeries*** 166 to *nil*. Thus the marker will snap to any series point but not to interpolated line values.

Events

Delphi: `property OnMarkerMoving: TNotifyEvent read mMarkerMoving write mMarkerMoving;`

C++: `__property Classes::TNotifyEvent OnMarkerMoving = {read=mMarkerMoving, write=mMarkerMoving};`

- This event is triggered during the dragging of the marker with the mouse when the position changes.

Delphi: `property OnMarkerMoved: TNotifyEvent read mMarkerMoved write mMarkerMoved;`

C++: `__property Classes::TNotifyEvent OnMarkerMoved = {read=mMarkerMoved, write=mMarkerMoved};`

- This event is triggered when the dragging of the marker with the mouse finishes.

Public Methods

Delphi:
```
procedure WriteTo(FileName: string); overload;
procedure WriteTo(Stream: TStream); overload; virtual;
procedure WriteTo(Node: IXMLNode); overload;
procedure WriteTo(Ini: TCustomIniFile; Section: string);
overload; virtual;
```

C++:
```
void __fastcall WriteTo(AnsiString FileName);
virtual void __fastcall WriteTo(Classes::TStream* Stream);
void __fastcall WriteTo(Xmlintf::_di_IXMLNode Node);
virtual void __fastcall WriteTo(Inifiles::TCustomIniFile*
Ini, AnsiString Section);
```

- The marker settings can be saved to a file in four different ways:
  to a binary file specified by name,
  to any binary stream,
  to an XML document node,
  to a specified section of an INI-file.

Delphi:
```
procedure ReadFrom(FileName: string); overload;
procedure ReadFrom(Stream: TStream); overload; virtual;
procedure ReadFrom(Node: IXMLNode); overload; virtual;
procedure ReadFrom(Ini: TCustomIniFile; Section: string);
overload; virtual;
```

C++:
```
void __fastcall ReadFrom(AnsiString FileName);
virtual void __fastcall ReadFrom(Classes::TStream* Stream);
virtual void __fastcall ReadFrom(Xmlintf::_di_IXMLNode Node);
virtual void __fastcall ReadFrom(Inifiles::TCustomIniFile*
Ini, AnsiString Section);
```

- The marker settings can be read from a file in four different ways:
  from a binary file specified by name,
  from any binary stream,
  from an XML document node,
  from a specified section of an INI file.

3.5.39 TRtVerticalMarker Class

This class derived from TRtMovable 163 provides a control displaying a vertical line that can be moved around horizontally in the graph. In addition to the parent properties, it publishes X 163, XAxis 163, Caption 164, CaptionVisible 164, CaptionVertical 164, CaptionDeltaX 164, CaptionDeltaY 164, CaptionPosition 164 and Line 164.

Unit/Namespace: `RtMarkers`

Declaration

Delphi: `TRtVerticalMarker = class(TRtMovable);`
C++: `class TRtVerticalMarker : public TRtMovable;`

3.5.40 TRtHorizontalMarker Class

This class derived from TRtMovable 163 provides a control displaying a horizontal line that can be moved around vertically in the graph. In addition to the parent properties, it publishes Y 164, YAxis 163, Caption 164, CaptionVisible 164, CaptionVertical 164, CaptionDeltaX 164, CaptionDeltaY 164, CaptionPosition 164 and Line 164.

Unit/Namespace: `RtMarkers`

Declaration

Delphi: `TRtHorizontaMarker = class(TRtMovable);`
C++: `class TRtHorizontaMarker : public TRtMovable;`

3.5.41 TRtCrossHair Class

This class derived from TRtMovable 163 provides a control displaying crossed horizontal and vertical lines that can be moved around in the graph. A caption can be displayed at the intersection. In addition to the parent properties, it publishes X 163, Y 164 and XAxis 163, YAxis 163, Line 164 and all the properties of the HorizontalMarker 169 and VertikalMarker 169 subcomponents.

Unit/Namespace: `RtMarkers`

Declaration

Delphi: `TRtCrossHair = class(TRtMovable);`
C++: `class TRtCrossHair : public TRtMovable;`

Published Properties

Delphi: `property Caption: TRtRichCaption 70 read GetCaption write SetCaption;`
C++: `__property WideString Caption = {read=GetCaption, write=SetCaption};`

- The optional caption displayed at the line intersection.

Delphi: `property CaptionDeltaX: Integer read FCaptionDeltaX write SetCaptionDeltaX;`
C++: `__property int CaptionDeltaX = {read=FCaptionDeltaX, write=SetCaptionDeltaX, nodefault};`
The X-offset of the caption in relation to the line intersection.

Delphi: `property CaptionDeltaY: Integer read FCaptionDeltaY write SetCaptionDeltaY;`
C++: `__property int CaptionDeltaY = {read=FCaptionDeltaY, write=SetCaptionDeltaY, nodefault};`
The Y-offset of the caption in relation to the line intersection.

Delphi:
```
TRtXHairCaptionPosition=(RightTop,RightBottom,LeftBottom,
LeftTop);
property CaptionPosition: TRtXHairCaptionPosition read
FCaptionPosition write SetCaptionPosition;
```
C++:
```
enum TRtXHairCaptionPosition { RightTop, RightBottom,
LeftBottom, LeftTop };
__property TRtXHairCaptionPosition CaptionPosition =
{read=FCaptionPosition, write=SetCaptionPosition, nodefault};
```
The position of the caption relative to the line intersection.

Delphi: `property HorizontalCursor: TCursor read GetHCursor write SetHCursor;`
C++: `__property Controls::TCursor HorizontalCursor = {read=GetHCursor, write=SetHCursor, nodefault};`
The mouse cursor shown with the horizontal line.

Delphi: `property VerticalCursor: TCursor read GetVCursor write SetVCursor;`
C++: `__property Controls::TCursor VerticalCursor = {read=GetVCursor, write=SetVCursor, nodefault};`
The mouse cursor shown with the vertical line.

Delphi: `property HorizontalMarker:TRtHorizontalMarker TCursor read FHHelper write FHHelper;`
C++: `__property TRtHorizontalMarker* HorizontalMarker = {read=FHHelper, write=FHHelper};`
The X-Hair internally is a combination of horizontal and vertical marker with a handler of the crossing area. This property gives access to the underlying TRtHorontalMarker 168 component.

Delphi: `property VerticalMarker:TRtHorizontalMarker TCursor read FVHelper write FVHelper;`
C++: `__property TRtVerticalMarker* VerticalMarker = {read=FVHelper, write=FVHelper};`
The X-Hair internally is a combination of horizontal and vertical marker with a handler of the crossing area. This property gives access to the underlying VerticalMarker 169 component.

3.5.42 TRtLabelWithArrowMarker Class

This class derived from TRtMovable 163 provides a control displaying an arrow with an optional bend in the line, leading up to a caption. The control can be moved around in the graph. The arrow and the caption position can be modified by the mouse and show floating grips. In addition to the parent properties, it publishes X 163, Y 164 and XAxis 163, YAxis 163.

Unit/Namespace: `RtMarkers`

Declaration

Delphi: `TRtLabelWithArrowMarker = class(TRtMovable);`
C++: `class TRtLabelWithArrowMarker : public TRtMovable;`

Published Properties

Delphi: `property Arrow: TRtArrowSettings read FArrowSettings write FArrowSettings;`
C++: `__property Rtstyles::TRtArrowSettings* Arrow = {read=FArrowSettings, write=FArrowSettings};`
The settings for the arrow line.

Delphi: `property ArrowAngle: Single read GetArrowAngle write SetArrowAngle;`
C++: `__property float ArrowAngle = {read=GetArrowAngle, write=SetArrowAngle};`
The angle in degrees at which the arrow is drawn.

Delphi: `property ArrowLength: Single read GetArrowLength write SetArrowLength;`
C++: `__property float ArrowLength = {read=GetArrowLength, write=SetArrowLength};`
The length up to the bend in the arrow line.

Delphi: `property ArrowDeltaX: Integer read FArrowDeltaX write SetArrowDeltaX;`
C++: `__property int ArrowDeltaX = {read=FArrowDeltaX, write=SetArrowDeltaX, nodefault};`
The X-offset of the end (bend) of the arrow line from the point.

Delphi: `property ArrowDeltaY: Integer read FArrowDeltaY write SetArrowDeltaY;`
C++: `__property int ArrowDeltaY = {read=FArrowDeltaY, write=SetArrowDeltaY, nodefault};`
The Y-offset of the end (bend) of the arrow line from the point.

Delphi: `property BentLength: Integer read FBentLength write SetBentLength;`
C++: `__property int BentLength = {read=FBentLength, write=SetBentLength, nodefault};`
The length of the bent part of the arrow line leading up to the caption.

Delphi: `property CaptionDeltaX: Integer read FCaptionDeltaX write SetCaptionDeltaX;`
C++: `__property int CaptionDeltaX = {read=FCaptionDeltaX, write=SetCaptionDeltaX, nodefault};`
The X-offset of the caption from the end of the arrow (bent part).

Delphi: `property CaptionDeltaY: Integer read FCaptionDeltaY write SetCaptionDeltaY;`
C++: `__property int CaptionDeltaY = {read=FCaptionDeltaY, write=SetCaptionDeltaY, nodefault};`
The Y-offset of the caption from the end of the arrow (bent part).

Delphi: `property HoverGrips: Boolean read FHoverGrips write FHoverGrips;`
C++: `__property bool HoverGrips = {read=FHoverGrips, write=FHoverGrips, nodefault};`

- If set to *true*, then the control will provide floating grips that can be dragged around with the mouse. This enables the user to alter the arrow length and angle, the size of the bent part and the position of the caption at runtime.

Delphi:
```
TRtArrowMarkerDraggingOption = (Hotspot, ArrowEnd, BentGrip, CaptionGrip);
TRtArrowMarkerDraggingOptions = set of TRtArrowMarkerDraggingOption;
const AllDraggingOptions = [Hotspot, ArrowEnd, BentGrip, CaptionGrip];
property DraggingOptions: TRtArrowMarkerDraggingOptions read FDraggingOptions write FDraggingOptions default AllDraggingOptions;
```
C++:
```
enum TRtArrowMarkerDraggingOption { Hotspot, ArrowEnd, BentGrip, CaptionGrip };
typedef Set<TRtArrowMarkerDraggingOption, Hotspot, CaptionGrip>  TRtArrowMarkerDraggingOptions;
__property TRtArrowMarkerDraggingOptions DraggingOptions = {read=FDraggingOptions, write=FDraggingOptions, default=15};
```

- This property allows to set the mouse dragging capabilities individually.
  Hotspot: controls the dragging of the control as total, meaning the position of the hotspot at the arrow tip.
  ArrowEnd: controls the dragging of the arrow end giving the size and angle of the arrow.
  BentGrip: controls the dragging of the arrow bent part length seizer grip.
  CaptionGrip: controls the dragging of the caption relative to the arrow.

Delphi: `property ArrowEndGripCursor: TCursor read GetArrowEndGripGripCursor write SetArrowEndGripGripCursor;`
C++: `__property Controls::TCursor ArrowEndGripCursor = {read=GetArrowEndGripGripCursor, write=SetArrowEndGripGripCursor, nodefault};`

- The mouse cursor shown at the floating grip at the end of the arrow.

Delphi: `property BentLengthGripCursor: TCursor read GetBentLengthGripCursor write SetBentLengthGripCursor;`
C++: `__property Controls::TCursor BentLengthGripCursor = {read=GetBentLengthGripCursor, write=SetBentLengthGripCursor, nodefault};`

- The mouse cursor shown at the floating grip at the end of the bent part of the arrow.

Delphi: `property CaptionGripCursor: TCursor read GetCaptionGripCursor write SetCaptionGripCursor;`
C++: `__property Controls::TCursor CaptionGripCursor = {read=GetCaptionGripCursor, write=SetCaptionGripCursor, nodefault};`

- The mouse cursor shown at the floating grip at the end of the caption.

Events

Delphi: `property OnArrowEndMoving: TNotifyEvent read mOnArrowEndMoving write mOnArrowEndMoving;`
C++: `__property Classes::TNotifyEvent OnArrowEndMoving = {read=mOnArrowEndMoving, write=mOnArrowEndMoving};`

This event is triggered during the dragging of the arrow end grip with the mouse when the position changes.

Delphi: `property OnArrowEndMoved: TNotifyEvent read mArrowEndMoved write mArrowEndMoved;`
C++: `__property Classes::TNotifyEvent OnArrowEndMoved = {read=mArrowEndMoved, write=mArrowEndMoved};`

This event is triggered when the dragging of the arrow end grip with the mouse finishes.

3.5.43 TRtFillSettings Class

This class defines the fill properties used to draw segments in pie charts.

Unit/Namespace: `RtPieDonut`

Declaration

Delphi: `TRtFillSettings = class(TPersistent)`
C++: `class TRtFillSettings : public Classes::TPersistent`

Public Properties

Delphi: `property ForeColor: TRtColor` 58 `read FForeColor write SetForeColor;`
C++: `__property Rtgdi::TRtColor ForeColor = {read=FForeColor, write=SetForeColor, nodefault};`

The color used for the foreground of the fill pattern.

Delphi: `property BackColor: TRtColor` 58 `read FBackColor write SetBackColor;`
C++: `__property Rtgdi::TRtColor BackColor = {read=FBackColor, write=SetBackColor, nodefault};`

The color used for the background of the fill pattern.

Delphi: `property AreaStyle: TRtAreaStyle` 59 `read FAreaStyle write SetAreaStyle;`
C++: `__property Rtgdi::TRtAreaStyle AreaStyle = {read=FAreaStyle, write=SetAreaStyle, nodefault};`

The area hatch style of the filled area.

3.5.44 TRtFillPalette Class

This class defines the list of fill properties used to draw segments in pie charts.

Unit/Namespace: `RtPieDonut`

Declaration

Delphi: `TRtFillPalette = class(TComponent)`
C++: `class TRtFillPalette : public Classes::TComponent`

Public Properties

Delphi: `property Count: Integer read GetCount;`
C++: `__property int Count = {read=FCount, nodefault};`
↳ The count of the fill settings stored.

Delphi: `property Items[i: Integer]: TRtFillSettings read GetItem write SetItem; default;`
C++: `TRtFillSettings* operator[](int i) { return Items[i]; } __property TRtFillSettings* Items[int i] = {read=GetItem, write=SetItem};`
↳ The array of fill settings objects.

Public Methods

Delphi: `function Add(Value: TRtFillSettings): Integer;`
C++: `int __fastcall Add(TRtFillSettings* Value);`
↳ Adds a fill settings object to the internal list. Returns the new count.

Delphi: `procedure Move(CurIndex, NewIndex: Integer);`
C++: `void __fastcall Move(int CurIndex, int NewIndex);`
↳ Moves an item from one position to another in the internal list.

Delphi: `procedure Insert(Index: Integer; Value: TRtFillSettings); reintroduce;`
C++: `HIDESBASE void __fastcall Insert(int Index, TRtFillSettings* Value);`
↳ Inserts a fill settings object at a defined position to the internal list.

Delphi: `procedure Delete(Index: Integer);`
C++: `void __fastcall Delete(int Index);`
↳ Deletes an item from the internal list at a defined position.

Delphi: `procedure Clear;`
C++: `void __fastcall Clear(void);`
↳ Clears the internal list of fill settings objects. Sets count to 0.

3.5.45 TRtPieChart Class

This class derived from TRtCustomGraph151 provides all the settings and methods related to drawing pie and donut charts.

Unit/Namespace: `RtPieDonut`

Declaration

Delphi: `TRtPieChart = class(TRtCustomGraph)`151
C++: `class TRtPieChart : public Rtgraph2d::TRtCustomGraph;`

Published Properties

Delphi: `property DistanceAtTop: Single read FDistanceAtTop write SetDistanceAtTop;`
C++: `__property float DistanceAtTop = {read=FDistanceBorderAxisTop, write=SetAtTop};`
↳ The minimum distance of the graph border to the top or the pie drawing area. The actual distance might be higher because the pie chart will automatically adjust to the center of the maximum width/height available.

Delphi: `property DistanceAtBottom: Single read FDistanceAtBottom write SetDistanceAtBottom;`
C++: `__property float DistanceAtBottom = {read=FDistanceBorderAxisBottom, write=SetAtBottom};`

The minimum distance of the graph border to the bottom or the pie drawing area. The actual distance might be higher because the pie chart will automatically adjust to the center of the maximum width/height available.

Delphi: `property DistanceAtLeft: Single read FDistanceAtLeft write SetDistanceAtLeft;`
C++: `__property float DistanceAtLeft = {read=FDistanceBorderAxisLeft, write=SetAtLeft};`

The minimum distance of the graph border to the Left or the pie drawing area. The actual distance might be higher because the pie chart will automatically adjust to the center of the maximum width/height available.

Delphi: `property DistanceAtRight: Single read FDistanceAtRight write SetDistanceAtRight;`
C++: `__property float DistanceAtRight = {read=FDistanceBorderAxisRight, write=SetAtRight};`

The minimum distance of the graph border to the Right or the pie drawing area. The actual distance might be higher because the pie chart will automatically adjust to the center of the maximum width/height available.

Delphi:
```
TRtPieChartType = (FlatPie,FlatDonut,ThreeDPie,ThreeDDonut);
property PieChartType: TRtPieChartType read FPieChartType
write SetPieChartType;
```
C++:
```
enum TRtPieChartType { FlatPie, FlatDonut, ThreeDPie,
ThreeDDonut };
__property TRtPieChartType PieChartType =
{read=FPieChartType, write=SetPieChartType, nodefault};
```

The display type of the chart. Flat pie, flat donut, 3D pie or 3D donut.

Delphi: `property StartAngle: Single read FStartAngle write SetStartAngle;`
C++: `__property float StartAngle = {read=FStartAngle, write=SetStartAngle};`

The angle in degrees used as start for the first segment.

Delphi: `property TotalAngle: Single read FTotalAngle write SetTotalAngle;`
C++: `__property float TotalAngle = {read=FTotalAngle, write=SetTotalAngle};`

The angle in degrees giving the sum of sweeps of all segments.

Delphi: `property Values: TRtCustomDoubleVector` 94 `read FValues write SetValues;`
C++: `__property Rtvectors::TRtCustomDoubleVector* Values = {read=FValues, write=SetValues};`

The reference to the storage vector of the data for calculating the sweeps of the pie segments.

Delphi: `property ExplodePercent: TRtCustomDoubleVector` 94 `read FExplodePercent write SetExplodePercent;`
C++: `__property Rtvectors::TRtCustomDoubleVector* ExplodePercent = {read=FExplodePercent, write=SetExplodePercent};`

The reference to the storage vector of the data for giving the percentage of radial shifts of the pie segments. Values higher than 100 are set to 100. Values lover 0 are set to 0.

Delphi: `property OrderAscending: Boolean read FOrderAscending write SetOrderAscending;`
C++: `__property bool OrderAscending = {read=FOrderAscending, write=SetOrderAscending, nodefault};`

If this property is set to *true*, the segments will be ordered starting with the smallest value.

Delphi: `property ThreeDElevation: Single read F3DElevation write Set3DElevation;`
C++: `__property float ThreeDElevation = {read=F3DElevation, write=Set3DElevation};`

The elevation in percent that the pie is higher than the width of the pie. This property is only used with ***PieChartType*** *ThreeDPie and ThreeDDonut*.

Delphi: `property ThreeDPieHeight: Single read F3DPieHeightPercent write Set3DPieHeight;`
C++: `__property float ThreeDPieHeight = {read=F3DPieHeightPercent, write=Set3DPieHeight};`

The height of the side of the pie in percent of the width of the pie. This property is only used with ***PieChartType*** *ThreeDPie and ThreeDDonut*.

Delphi: `property DonutHoleSize: Single read FDonutHoleSize write SetDonutHoleSize;`
C++: `__property float DonutHoleSize = {read=FDonutHoleSize, write=SetDonutHoleSize};`

The size of the hole of the donut in percent of the width of the donut. This property is only used with ***PieChartType*** *FlatDonut and ThreeDDonut*.

Delphi: `property SegmentSettingsPalette: TRtFillPalette` 172 `read FSegmentSettingsPalette write SetSegmentSettingsPalette;`
C++: `__property Rtstyles::TRtLineSettings* SegmentBorder = {read=FSegmentBorder, write=SetSegmentBorder};`

The reference to the fill settings palette used to draw the segments fills.

Delphi: `property SegmentBorder: TRtLineSettings` 99 `read FSegmentBorder write SetSegmentBorder;`
C++: `__property Rtstyles::TRtLineSettings* SegmentBorder = {read=FSegmentBorder, write=SetSegmentBorder};`

The settings used to draw the border line of the segments.

Delphi: `property DropShadow: Boolean read FDropShadow write SetDropShadow;`
C++: `__property bool DropShadow = {read=FDropShadow, write=SetDropShadow, nodefault};`

If this property is set to *true* the pie/donut will be drawn with a shadow to the background.

Delphi: `property Font: TFont read FFont write SetFont;`
C++: `__property Graphics::TFont* Font = {read=FFont, write=SetFont};`

The font used for the optional labels.

Delphi: `property LabelColor: TRtColor read FLabelColor write SetLabelColor;`
C++: `__property Rtgdi::TRtColor LabelColor = {read=FLabelColor, write=SetLabelColor, nodefault};`

The color of the optional labels.

Delphi: `property float LabelSize = {read=FLabelSize, write=SetLabelSize};`
C++: `__property float LabelSize = {read=FLabelSize, write=SetLabelSize};`

The font size of the optional labels as a percent of the pie width. The class does not use absolute sizes: this ensures that the chart looks the same when resized.

Delphi:
```
TRtPieLabelPosition = (Invisible, InsideHorizontal,
InsideAligned, Callouts);
property LabelPosition: TRtPieLabelPosition read
FLabelPosition write SetLabelPosition;
```
C++:
```
enum TRtPieLabelPosition { Invisible, InsideHorizontal,
InsideAligned, Callouts };
__property TRtPieLabelPosition LabelPosition =
{read=FLabelPosition, write=SetLabelPosition, nodefault};
```

The drawing position and angle of the labels. Totally invisible, inside the segments with horizontal text, inside the segments with text angle following the center of the segment and outside the pie as horizontal callouts.

Delphi:
```
TRtLabelStyle=(Value, CaptionsList, Percent, CaptionAndValue,
CaptionAndPercent);
property LabelFrom: TRtLabelStyle read FLabelFrom write
SetLabelFrom;
```
C++:
```
enum TRtLabelStyle { Value, CaptionsList, Percent,
CaptionAndValue, CaptionAndPercent };
__property Rtbars::TRtLabelStyle LabelFrom =
{read=FLabelFrom, write=SetLabelFrom, nodefault};
```

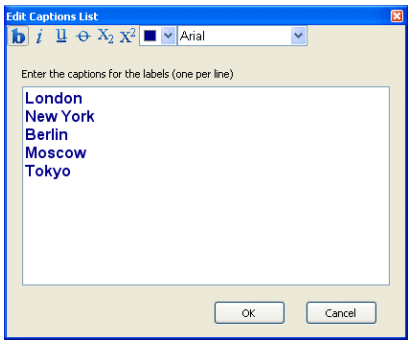
If *Value* has been set, the label displayed will show the numerical value of the segment. The display can be altered using the LabelFormat 176 property. If *CaptionsList* is set, the label retrieves the text from the Captions 177 string list. If *Percent* is selected the text will display the value as percent of the sum of values using the optional LabelFormat. CaptionAndValue and CaptionAndPercent will display combination of both.

Delphi: `property LabelFormat: string read FLabelFormat write SetLabelFormat;`
C++: `__property AnsiString LabelFormat = {read=FLabelFormat, write=SetLabelFormat};`

If LabelFrom 176 has been set to *Value* or *Percent*, the label displayed will show the numerical value of the segment. The format is used in the same way as with the ***FormatFloat*** function. A value "1.234" with a format of "0.00" will therefore display "1.23". An empty format string will display all significant digits.

Delphi: `property Captions: TRtCaptions read GetCaptionsList write SetCaptionsList;`
C++: `__property Tntclasses::TTntStrings* Captions = {read=GetCaptionsList, write=SetCaptionsList};`

If LabelFrom has been set to *CaptionsList*, the labels displayed at the segments retrieve their text from a string list.



A property editor is provided for editing all the captions, available when clicking the ellipsis button in the object inspector with enhanced styles.

Delphi: `property DataSource: TDataSource read GetDataSource write SetDataSource;`
C++: `__property Db::TDataSource* DataSource = {read=GetDataSource, write=SetDataSource};`

Data source to connect to, providing the database supplying the captions for the optional labels. Leave empty if you do not want to link to a database.

Delphi: `property CaptionsField: string read GetDataField write SetDataField;`
C++: `__property AnsiString CaptionsField = {read=GetDataField, write=SetDataField};`

Name of the database field used to get the captions for the optional labels. Leave empty if you do not want to link to a database.

Delphi: `property CalloutArrow: TRtArrowSettings read FCalloutArrow write SetCalloutArrow;`
C++: `__property Rtstyles::TRtArrowSettings* CalloutArrow = {read=FCalloutArrow, write=SetCalloutArrow};`

The settings for the arrow to be drawn from the callout to the segment border. This line is only visible if CalloutArrowVisibleMode is set to *AllVisible* or if set to *VisibleIfOverlapped* and the callouts would have overlapped and therefore have been shifted up/downwards. Callouts are only drawn if ***LabelPosition*** = *Callouts*.

Delphi:
```
TRtCalloutArrowVisibleMode = (None, VisibleIfOverlapped, AllVisible);
property CalloutArrowVisibleMode: TRtCalloutArrowVisibleMode read FCalloutArrowVisibleMode write SetCalloutArrowVisibleMode;
```
C++:
```
enum TRtCalloutArrowVisibleMode { None, VisibleIfOverlapped, AllVisible };
__property TRtCalloutArrowVisibleMode CalloutArrowVisibleMode = {read=FCalloutArrowVisibleMode, write=SetCalloutArrowVisibleMode, nodefault};
```

- This property controls the display behaviour of optional arrows to be drawn from the callouts to the segments borders. The callouts positions are calculated to have a minimum distance to the pie specified with the CalloutDistanceToPie property. If the callouts would overlap at this positions the most outer callout is shifted further up/ down to meet the condition that the overlapping callouts will have the ***CalloutDistanceToPie***. You can set the arrows to be never drawn, drawn only if there has been an overlapping callout or in any case. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutBackColor: TRtColor` 58 `read FCalloutBackColor write SetCalloutBackColor;`

C++: `__property Rtgdi::TRtColor CalloutBackColor = {read=FCalloutBackColor, write=SetCalloutBackColor, nodefault};`

- The color of the background used to draw the callout if CalloutGradient 178.Visible = *false*. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutGradient: TRtGradientSettings` 64 `read FCalloutGradient write SetCalloutGradient;`

C++: `__property Rtstyles::TRtGradientSettings* CalloutGradient = {read=FCalloutGradient, write=SetCalloutGradient};`

- Settings for the optional gradient to be drawn as background. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutFrame: TRtLineSettings` 99 `read FCalloutFrame write SetCalloutFrame;`

C++: `__property Rtstyles::TRtLineSettings* CalloutFrame = {read=FCalloutFrame, write=SetCalloutFrame};`

- Settings for the drawing of a border frame around the callout. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutRaised: Boolean read FCalloutRaised write SetCalloutRaised;`

C++: `__property bool CalloutRaised = {read=FCalloutRaised, write=SetCalloutRaised, nodefault};`

- If this property is set to *true* the callout will be drawn with a raised border. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutLowered: Boolean read FCalloutLowered write SetCalloutLowered;`

C++: `__property bool CalloutLowered = {read=FCalloutLowered, write=SetCalloutLowered, nodefault};`

- If this property is set to *true* the callout will be drawn with a lowered border. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutShadow: Boolean read FCalloutShadow write SetCalloutShadow;`

C++: `__property bool CalloutShadow = {read=FCalloutShadow, write=SetCalloutShadow, nodefault};`

- If this property is set to *true* the callout will be drawn with a shadow to the background. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutDistanceToPie: Single read FCalloutDistanceToPie write SetCalloutDistanceToPie;`
C++: `__property float CalloutDistanceToPie = {read=FCalloutDistanceToPie, write=SetCalloutDistanceToPie};`

The minimum distance to the callout keeps from the pie border in percent of the label font size 176. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutSpaceToBorder: Single read FCalloutSpaceToBorder write SetCalloutSpaceToBorder;`
C++: `__property float CalloutSpaceToBorder = {read=FCalloutSpaceToBorder, write=SetCalloutSpaceToBorder};`

The distance from the text to the callout border in percent of the label font size 176. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property CalloutRounding: Single read FCalloutRounding write SetCalloutRounding;`
C++: `__property float CalloutRounding = {read=FCalloutRounding, write=SetCalloutRounding};`

The radius used to draw the callout border round corners in percent of the label font size 176.. Callouts are only drawn if ***LabelPosition*** 176 = *Callouts*.

Delphi: `property OutliersVisible: Boolean read FOutliersVisible write SetOutliersVisible;`
C++: `__property bool OutliersVisible = {read=FOutliersVisible, write=SetOutliersVisible, nodefault};`

If set to *false*, the pie chart will not display any data points as segments marked as outliers 95. If set *true*, all data available will be used for display.

Delphi: `property AutoUpdate: Boolean read FAutoUpdate write FAutoUpdate;`
C++: `__property bool AutoUpdate = {read=FAutoUpdate, write=FAutoUpdate, nodefault};`

If set to *true*, the chart will be automatically updated when a value is added, changed or removed via the Values- or ExplodePercent vectors.

Events

Delphi: `property ValueTransformation: TRtValueTransformation` 98 `read mValueTransformation write SetValueTransformation;`
C++: `__property TRtValueTransformation ValueTransformation = {read=mValueTransformation, write=SetValueTransformation};`

This event is triggered each time a Value is needed for calculation. You can use it to return any other transformed value instead.

Delphi: `property ExplodePercentTransformation: TRtValueTransformation` 98 `read mExplodePercentTransformation write SetExplodePercentTransformation;`
C++: `__property TRtValueTransformation ExplodePercentTransformation = {read=mExplodePercentTransformation, write=SetExplodePercentTransformation};`

This event is triggered each time a explosion shift percent Value is needed for calculation. It is also called if you did not specify any vector object for ExplodePercent 175. You can use it to return any other transformed value instead.

Event

Delphi:
```
TRtGetLabelEventArgs=record
  Caption: TRtRichCaption;
  Index: Integer;
end;
TRtGetLabelEventHandler = procedure(Sender: TObject; var e:
TRtGetLabelEventArgs) of object;
property OnGetLabel: TRtGetLabelEventHandler read mGetLabel
write mGetLabel;
```

C++:
```
struct TRtGetLabelEventArgs{
public:
      WideString Caption;
      int Index;
};
typedef void __fastcall (__closure *TRtBarLabelEventHandler)
(System::TObject* Sender, TRtGetLabelEventArgs &e);
__property TRtGetLabelEventHandler OnGetLabel =
{read=mGetLabel, write=mGetLabel};
```

This event is triggered each time an optional label caption is required. You can use this to implement your own formatting, for example. The parameter passed gives access to the default caption and the index of the current segment. You can alter the ***e.Caption*** to your needs.

Delphi:
```
TRtPenEventArgs = record
  Index: Integer;
  X,Y: Double;
  Pen: TRtPen 58;
  Visible: Boolean;
end;
TRtPenEventHandler = procedure(Sender: TObject; var e:
TRtPenEventArgs) of object;
property OnGetPen: TRtPenEventHandler read mGetPen write
mGetPen;
```

C++:
```
struct TRtPenEventArgs{
public:
      int Index;
      double X;
      double Y;
      Rtgdi::TRtPen* Pen;
      bool Visible;
};
typedef void __fastcall (__closure *TRtPenEventHandler)
(System::TObject* Sender, TRtPenEventArgs &e);
__property TRtPenEventHandler OnGetPen = {read=mGetPen,
write=mGetPen};
```

This event can be used to customize the style of single line segments in the pie chart. If specified, it will be called when drawing the line for each segment. X will contain the Value representing the segment Y will contain the sweep angle of the segment.

Delphi:
```
TRtBrushEventArgs = record
  Index: Integer;
  X,Y: Double;
  Brush: TRtBrush 58;
  Visible: Boolean;
end;
```

```
TRtBrushEventHandler = procedure( Sender: TObject; var e:
TRtBrushEventArgs) of object;
property OnGetBrush: TRtBrushEventHandler read mGetBrush
write mGetBrush;
```

C++:
```
struct TRtBrushEventArgs{
public:
      int Index;
      double X;
      double Y;
      Gdipobj::TGPBrush* Brush;
      bool Visible;
};
typedef void __fastcall ( __closure *TRtBrushEventHandler)
( System::TObject* Sender, TRtBrushEventArgs &e);
__property TRtBrushEventHandler OnGetBrush = {read=mGetBrush,
write=mGetBrush};
```

This event can be used to customize the style of single area below the line segments in the pie chart. If specified, it will be called when filling the area on each segment. X will contain the Value representing the segment Y will contain the sweep angle of the segment.

3.5.46 TRtPieLegend Class

This class derived from TRtCustomLegend 131 provides all the settings and methods related to a legend for a pie chart.

Unit/Namespace: `RtPieLegend`

Declaration

Delphi:
```
TRtPieLegend = class( TRtCustomLegend 131)
```

C++:
```
class TRtPieLegend : public Rtlegend::TRtCustomLegend
```

Published Properties

Delphi:
```
property PieChart: TRtPieChart 173 read FPieChart write
SetPieChart;
```

C++:
```
__property Rtpiedonut::TRtPieChart* PieChart =
{read=FPieChart, write=SetPieChart};
```

The assigned pie chart containing the segment items to display. At design time it is set automatically to the first chart you place the control to.

Delphi:
```
property DistanceToPie: Single read FDistanceToPie write
SetDistanceToPie;
```

C++:
```
__property float DistanceToPie = {read=FDistanceToPie,
write=SetDistanceToPie};
```

If the legend is placed at a pie chart as parent this property will define the space in pixels the legend will be kept away from the pie and its callouts.

Delphi: `TRtPieLegendPosition=( LeftTop, LeftCenter, LeftBottom, CenterTop, CenterBottom, RightTop, RightCenter, RightBottom);`
`property Position: TRtPieLegendPosition read FPosition write SetPosition;`

C++: `enum TRtPieLegendPosition { LeftTop, LeftCenter, LeftBottom, CenterTop, CenterBottom, RightTop, RightCenter, RightBottom };`
`__property TRtPieLegendPosition Position = {read=FPosition, write=SetPosition, nodefault};`

The position of the legend if placed to a pie chart as parent.

Delphi: `TRtLabelStyle=( Value, CaptionsList, Percent, CaptionAndValue, CaptionAndPercent);`
`property LabelsFrom: TRtLabelStyle read FLabelsFrom write SetLabelsFrom;`

C++: `enum TRtLabelStyle { Value, CaptionsList, Percent, CaptionAndValue, CaptionAndPercent };`
`__property Rtbars::TRtLabelStyle LabelsFrom = {read=FLabelsFrom, write=SetLabelsFrom, nodefault};`

If *Value* has been set, the label displayed will show the numerical value of the segment. The display can be altered using the LabelFormat 176 property. If *CaptionsList* is set, the label retrieves the text from the Captions 177 string list. If *Percent* is selected the text will display the value as percent of the sum of values using the optional LabelFormat. CaptionAndValue and CaptionAndPercent will display combination of both.

Delphi: `property LabelFormat: string read FLabelFormat write SetLabelFormat;`

C++: `__property AnsiString LabelFormat = {read=FLabelFormat, write=SetLabelFormat};`

If LabelFrom 176 has been set to *Value* or *Percent*, the label displayed will show the numerical value of the segment. The format is used in the same way as with the ***FormatFloat*** function. A value "1.234" with a format of "0.00" will therefore display "1.23". An empty format string will display all significant digits.

3.5.47 TRtPieSettingsTool Class

This class provides a tool window for setting-up all the properties of the pie chart, the fill styles, the labels and the legend.

Unit/Namespace: `RtPieDonut`

Declaration

Delphi: `TRtPieSettingsTool = class( TComponent)`

C++: `class TRtPieSettingsTool : public Classes::TComponent;`

Published Properties

Delphi: `property PieChart: TRtPieChart read FPieChart write SetPieChart;`

C++: `__property Rtpiedonut::TRtPieChart* PieChart = {read=FPieChart, write=SetPieChart};`

The assigned pie chart containing the properties to be set.

Delphi: `property Caption: string read FCaption write SetCaption;`
C++: `__property AnsiString Caption = {read=FCaption, write=SetCaption};`
The caption used in the title of the tool window.

3.6 Frames

The following frame classes have been provided to enable you to built up the user interface of your program in an easy way. It might be not useful to you to give the users access to all the options available using the TRtGraphSettingsTool 161 and TRtPieSettingsTool 182 dialogs. It also might not be feasible to use a single always stay on top window to set all the properties. The other extreme would be to built up a special user interface e.g. using the controls in the standard palette 8. The frames now give a possibility to generate a user interface which is more customized but with a pre built functionality to set up all the properties of the related graph controls.

3.6.1 TRtGradientFrame Class

This class provides a tool frame for setting-up all the properties of a gradient except the visible property, which can be implemented easily by an additional checkbox. This frame is used internally by nearly all pages of the TRtGraphSettingsTool 161 and TRtPieSettingsTool 182 dialogs.

Unit/Namespace: `RtGradientFrm`

Declaration

Delphi: `TRtGradientFrame = class(TFrame)`
C++: `class TRtGradientFrame : public Forms::TFrame`

Public Property

Delphi: `property Gradient: TRtGradientSettings read FGradient write SetGradient;`
C++: `__property Rtstyles::TRtGradientSettings* Gradient = {read=FGradient, write=SetGradient};`
The assigned gradient containing the properties to be set.

Event

Delphi: `property OnChanged: TNotifyEvent read mChanged write mChanged;`
C++: `__property Classes::TNotifyEvent OnChanged = {read=mChanged, write=mChanged};`
This event is triggered each time a property was changed. It can be used to update any additional controls e.g. custom draw lists.

3.6.2 TRtGraphSetingsFrame Class

This class provides a tool frame for setting-up all the properties of a Cartesian graph control. This frame is used internally by the graph pages of the TRtGraphSettingsTool 161.

Unit/Namespace: `RtGraphSettingsFrm`

Declaration

Delphi: `TRtGraphSettingsFrame = class(TFrame)`
C++: `class TRtGraphSettingsFrame : public Forms::TFrame`

Published Properties

Delphi: `property Graph: TRtGraph2D read FGraph write SetGraph;`
C++: `__property Rtgraph2d::TRtGraph2D* Graph = {read=FGraph, write=SetGraph};`

↳ The assigned graph containing the properties to be set.

Delphi: `property ZoomScrollPanningOptions: Boolean read FZoomScrollPanningOptions write SetZoomScrollPanningOptions;`
C++: `__property bool ZoomScrollPanningOptions = {read=FZoomScrollPanningOptions, write=SetZoomScrollPanningOptions, nodefault};`

↳ If set to *true*, then the frame will allow the user to set up the mouse and scroll options for auto zooming and panning.

3.6.3 TRtAxisSettingsFrame Class

This class provides a tool frame for setting-up all the properties of an axis. This frame is used internally by the lower part of the axis page of the TRtGraphSettingsTool 161.

Unit/Namespace: `RtAxisSettingsFrm`

Declaration

Delphi: `TRtAxisSettingsFrame = class(TFrame)`
C++: `class TRtAxisSettingsFrame : public Forms::TFrame`

Published Properties

Delphi: `property SelectedAxis: TRtAxis` 138 `read FSelectedAxis write SetSelectedAxis;`
C++: `__property Rtaxis::TRtAxis* SelectedAxis = {read=FSelectedAxis, write=SetSelectedAxis};`

↳ The active axis giving the states of the controls. The options set within the frame will be applied on this axis only if no ***RelatedAxisList*** is specified.

Delphi: `property RelatedAxesList: TCustomListBox read FAxesList write SetRelatedAxesList;`
C++: `__property Stdctrls::TCustomListBox* RelatedAxesList = {read=FAxesList, write=SetRelatedAxesList};`

↳ The optional related list box containing the axes. This list can be used to give back a multiple selection of axes objects contained in the lists ***Items.Objects***. If you leave the property as *nil* the changes will be applied on the SelectedAxis only.

Delphi: `property ExtendedAxisOptions: Boolean read FExtendedAxisOptions write SetExtendedAxisOptions;`
C++: `__property bool ExtendedAxisOptions = {read=FExtendedAxisOptions, write=SetExtendedAxisOptions, nodefault};`

↳ If set to *true*, then the tool window will allow the user to set up additional "enhanced" options, such as separate auto end settings for start and stop, end rounding selection, slave axis settings and moving slit options.

3.6.4 TRtSeriesSettingsFrame Class

This class provides a tool frame for setting-up all the properties of a series. It doesn't matter which type the series has. The control will update automatically displaying the options relevant to the specific type of the selected series. This frame is used internally by the series page of the TRtGraphSettingsTool 161 dialog.

Unit/Namespace: `RtSeriesSettingsFrm`

Declaration

Delphi: `TRtSeriesSettingsFrame = class(TFrame)`
C++: `class TRtSeriesSettingsFrame : public Forms::TFrame`

Published Properties

Delphi: `property SelectedSeries: TRtSeries 103 read FSelectedSeries write SetSelectedSeries;`
C++: `__property Rtseries::TRtSeries* SelectedSeries = {read=FSelectedSeries, write=SetSelectedSeries};`

The assigned series containing the properties to be set.

Delphi: `property RelatedSeriesList: TListBox read FSeriesList write SetRelatedSeriesList;`
C++: `__property Stdctrls::TListBox* RelatedSeriesList = {read=FSeriesList, write=SetRelatedSeriesList};`

The optional related list box containing the series. This is used to update the list display if any relevant property was changed. You can keep this property *nil* if you don't use a list to select or nothing needs to be updated.

Delphi: `property ExtendedBarLabelOptions: Boolean read FExtendedBarLabelOptions write SetExtendedBarLabelOptions;`
C++: `__property bool ExtendedBarLabelOptions = {read=FExtendedBarLabelOptions, write=SetExtendedBarLabelOptions, nodefault};`

If set to *true*, then the tool frame will allow the user to set up additional "expert" options, such as selecting whether the labels are set by the values or the captions list and also the number format for the labels.

Delphi: `property InteractiveCalculationRanges: Boolean read FInteractiveCalculationRanges write SetInteractiveCalculationRanges;`
C++: `__property bool InteractiveCalculationRanges = {read=FInteractiveCalculationRanges, write=SetInteractiveCalculationRanges, nodefault};`

If set to *true*, then the tool frame will allow the user to set up calculation and display ranges interactively for function series 119. Two vertical markers can be dragged around the graph to define new ranges. If a range has been changed the corresponding AutoStart 120/AutoStop 120 or DisplayAutoStart 120/DisplayAutoStop 121 properties will be set to *false*. If this option is set to *false*, check boxes give access to the Auto... properties.

Delphi: `property ShowFitCovariance: Boolean read FShowFitCovariance write SetShowFitCovariance;`
C++: `__property bool ShowFitCovariance = {read=FShowFitCovariance, write=SetShowFitCovariance, nodefault};`

If set to *true*, then the tool frame will allow the user to set up DoCovariance 126 on TRtFittedLine 125 series.

Delphi: `property bool ShowOutliersSupport = {read=FShowOutliersSupport, write=SetShowOutliersSupport, nodefault};`
C++: `__property bool ShowOutliersSupport = {read=F ShowOutliersSupport, write=SetShowOutliersSupport, nodefault};`

If set to *true*, then the tool frame will allow the user to set up whether outliers are displayed or hidden in series options.

3.6.5 TRtLegendSettingsFrame Class

This class provides a tool frame for setting-up all the properties of a legend for the Cartesian graph control. This frame is used internally by the legend pages of the TRtGraphSettingsTool 161 dialog.

Unit/Namespace: `RtLegendSettingsFrm`

Declaration

Delphi: `TRtLegendSettingsFrame = class(TFrame)`
C++: `class TRtLegendSettingsFrame : public Forms::TFrame`

Published Property

Delphi: `property Legend: TRtLegend read FLegend write SetLegend;`
C++: `__property Rtlegend::TRtLegend* Legend = {read=FLegend, write=SetLegend};`

The assigned legend containing the properties to be set.

3.6.6 TRtPieGeneralSettingsFrame Class

This class provides a tool frame for setting-up all the general properties of a pie chart. This frame is used internally by the first page of the TRtPieSettingsTool 182.

Unit/Namespace: `RtGradientFrm`

Declaration

Delphi: `TRtPieGeneralSettingsFrame = class(TFrame)`
C++: `class TRtPieGeneralSettingsFrame : public Forms::TFrame`

Published Property

Delphi: `property PieChart: TRtPieChart 173 read FPieChart write SetPieChart;`
C++: `__property Rtpiedonut::TRtPieChart* PieChart = {read=FPieChart, write=SetPieChart};`

The assigned pie/donut chart containing the properties to be set.

3.6.7 TRtFillPaletteFrame Class

This class provides a tool frame for setting-up all the fill settings of a TRtFillPalette 172 component. This frame is used internally by the segments group of the second page of the TRtPieSettingsTool 182.

Unit/Namespace: `RtFillPaletteFrm`

Declaration

Delphi: `TRtFillPaletteFrame = class(TFrame)`
C++: `class TRtFillPaletteFrame : public Forms::TFrame`

Published Property

Delphi: `property Palette: TRtFillPalette read FFillPalette write SetFillPalette;`
C++: `__property Rtpiedonut::TRtFillPalette* Palette = {read=FFillPalette, write=SetFillPalette};`
↳ The assigned palette containing the fill settings to be set.

3.6.8 TRtPieDonutFillFrame Class

This class provides a tool frame for setting-up the segment properties of a pie chart. This frame is used internally by the second page of the TRtPieSettingsTool 182.

Unit/Namespace: `RtPieDonutFillFrm`

Declaration

Delphi: `TRtPieDonutFillFrame = class(TFrame)`
C++: `class TRtPieDonutFillFrame : public Forms::TFrame`

Published Property

Delphi: `property PieChart: TRtPieChart 173 read FPieChart write SetPieChart;`
C++: `__property Rtpiedonut::TRtPieChart* PieChart = {read=FPieChart, write=SetPieChart};`
↳ The assigned pie/donut chart containing the properties to be set.

3.6.9 TRtPieLabelsFrame Class

This class provides a tool frame for setting-up all the label relevant properties of a pie chart. This frame is used internally by the third page of the TRtPieSettingsTool 182.

Unit/Namespace: `RtPieLabelsFrm`

Declaration

Delphi: `TRtPieLabelsFrame = class(TFrame)`
C++: `class TRtPieLabelsFrame : public Forms::TFrame`

Published Property

Delphi: `property PieChart: TRtPieChart 173 read FPieChart write SetPieChart;`
C++: `__property Rtpiedonut::TRtPieChart* PieChart = {read=FPieChart, write=SetPieChart};`
↳ The assigned pie/donut chart containing the properties to be set.

3.6.10 TRtPieLegendSettingsFrame Class

This class provides a tool frame for setting-up all the properties of a legend for a pie chart. This frame is used internally by the fourth page of the TRtPieSettingsTool 182.

Unit/Namespace: `RtGradientFrm`

Declaration

Delphi: `TRtPieLegendSettingsFrame = class(TFrame)`
C++: `class TRtPieLegendSettingsFrame : public Forms::TFrame`

Published Property

Delphi: `property Legend: TRtPieLegend read FLegend write SetLegend;`
C++: `__property Rtpielegend::TRtPieLegend* Legend = {read=FLegend, write=SetLegend};`
↳ The assigned pie/donut chart legend containing the properties to be set.

3.6.11 TRtSelectSeriesFrame Class

This class provides a checked list box containing the series of a graph. The visibility of the series can be controlled with the check boxes besides the series item drawing and the series caption.

Unit/Namespace: `RtSelectSeriesFrm`

Declaration

Delphi: `TRtSelectSeriesFrm = class(TFrame)`
C++: `class TRtSelectSeriesFrm : public Forms::TFrame`

Published Properties

Delphi: `property Graph: TRtGraph2D read FGraph write SetGraph;`
C++: `__property Rtgraph2d::TRtGraph2D* Graph = {read=FGraph, write=SetGraph};`
↳ The assigned graph containing the series visibility to be set.

Delphi: `property FromIncludeInLegend: Boolean read FFromIncludeInLegend write SetFromIncludeInLegend;`
C++: `__property bool FromIncludeInLegend = {read=FFromIncludeInLegend, write=SetFromIncludeInLegend, nodefault};`
↳ If set to *true*, then the list will show the list as it would display in a legend. Series merged in the display of the legend using the TRtSeries.MergeLegendItemWith 105 will be set in one go. If set to *false* the selection will be separate on each series.

Delphi: `property Font: TRtFont` 66 `read FFont write SetFont stored GetStoreFont;`
C++: `__property Rtrichlabel::TRtFont* Font = {read=FFont, write=SetFont, stored=GetStoreFont};`
↳ The font with enhanced settings used for the drawing the captions.

Delphi: `property ItemWidth: Integer read FItemWidth write SetItemWidth;`
C++: `__property int ItemWidth = {read=FItemWidth, write=SetItemWidth, nodefault};`
↳ The width of the items.

Delphi: `property ItemHeight: Integer read GetItemHeight write SetItemHeight;`
C++: `__property int ItemHeight = {read=GetItemHeight, write=SetItemHeight, nodefault};`
↳ The height of the items.

Index

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- H -

- N -

- O -

- P -

- R -

- S -

- T -

- U -

- V -

- W -

- X -

- Y -

- Z -