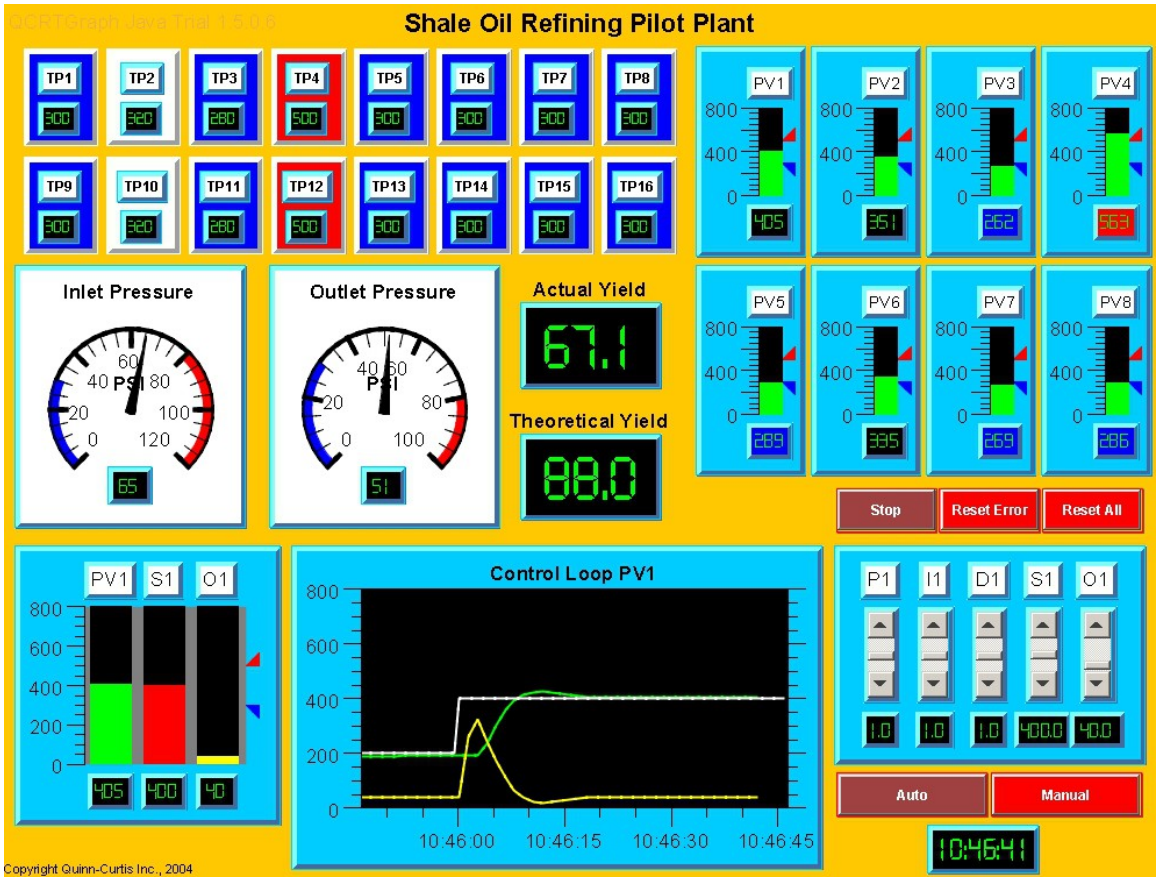
QCRTGraph Real-Time Graphics Tools for Java



Contact Information

Company Web Site: **http://**www.quinn-curtis.com

Electronic mail

General Information: info@quinn-curtis.com
Sales: sales@quinn-curtis.com

Technical Support Forum

http://www.quinn-curtis.com/ForumFrame.htm

Revision Date 9/20/2017 Rev. 3.0

Quinn-Curtis, Inc. Tools Tools for *Java* END-USER LICENSE AGREEMENT

IMPORTANT-READ CAREFULLY: This Software End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Quinn-Curtis, Inc. for the Quinn-Curtis, Inc. SOFTWARE identified above, which includes all Quinn-Curtis, Inc ***Java*** software (on any media) and related documentation (on any media). 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. If the SOFTWARE was mailed to you, return the media envelope, UNOPENED, along with the rest of the package to the location where you obtained it within 30 days from purchase.

1. The SOFTWARE is licensed, not sold.

2. GRANT OF LICENSE.

(A) Developer License. After you have purchased the license for SOFTWARE, and have received the file containing the licensed copy, you are licensed to copy the SOFTWARE only into the memory of the number of computers corresponding to the number of licenses purchased. The primary user of the computer on which each licensed copy of the SOFTWARE is installed may make a second copy for his or her exclusive use on a portable 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 license fee. You may not duplicate the SOFTWARE in whole or in part, except that you may make one copy of the SOFTWARE for backup or archival purposes. You may terminate this license at any time by destroying the original and all copies of the SOFTWARE in whatever form.

(B) 30-Day Trial License. You may download and use the SOFTWARE without charge on an evaluation basis for thirty (30) days from the day that you DOWNLOAD the trial version of the SOFTWARE. The termination date of the trial SOFTWARE is embedded in the downloaded SOFTWARE and cannot be changed. You must pay the license fee for a Developer License of the SOFTWARE to continue to use the SOFTWARE after the thirty (30) days. If you continue to use the SOFTWARE after the thirty (30) days without paying the license fee you will be using the SOFTWARE on an unlicensed basis.

Redistribution of 30-Day Trial Copy. Bear in mind that the 30-Day Trial version of the SOFTWARE becomes invalid 30-days after downloaded from our web site, or one of our sponsor's web sites. If you wish to redistribute the 30-day trial version of the SOFTWARE you should arrange to have it redistributed directly from our web site If you are using SOFTWARE on an evaluation basis you may make copies of the evaluation SOFTWARE as you wish; give exact copies of the original evaluation SOFTWARE to anyone; and distribute the evaluation SOFTWARE in its unmodified form via electronic means (Internet, BBS's, Shareware distribution libraries, CD-ROMs, etc.). You may not charge any fee for the copy or use of the evaluation SOFTWARE itself. You must not represent in any way that you are selling the SOFTWARE itself. You must distribute a copy of this EULA with any copy of the SOFTWARE and anyone to whom you distribute the SOFTWARE is subject to this EULA.

(C) Redistributable License. The standard Developer License permits the programmer to deploy and/or distribute applications that use the Quinn-Curtis SOFTWARE, royalty free. We cannot allow developers to use this SOFTWARE to create a graphics toolkit (a library or any type of graphics component that will be used in combination with a program development environment) for resale to other developers.

If you utilize the SOFTWARE in an application program, or in a web site deployment, should we ask, you must supply Quinn-Curtis, Inc. with the name of the application program and/or the URL where the SOFTWARE is installed and being used.

3. RESTRICTIONS. You may not reverse engineer, de-compile, or disassemble the SOFTWARE, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this

limitation. You may not rent, lease, or lend the SOFTWARE. You may not use the SOFTWARE to perform any illegal purpose.

4. SUPPORT SERVICES. Quinn-Curtis, Inc. may provide you with support services related to the SOFTWARE. Use of Support Services is governed by the Quinn-Curtis, Inc. polices and programs described in the user manual, in online documentation, and/or other Quinn-Curtis, Inc.-provided materials, as they may be modified from time to time. Any supplemental SOFTWARE code provided to you as part of the Support Services shall be considered part of the SOFTWARE and subject to the terms and conditions of this EULA. With respect to technical information you provide to Quinn-Curtis, Inc. as part of the Support Services, Quinn-Curtis, Inc. may use such information for its business purposes, including for product support and development. Quinn-Curtis, Inc. will not utilize such technical information in a form that personally identifies you.

5. TERMINATION. Without prejudice to any other rights, Quinn-Curtis, Inc. 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.

6. COPYRIGHT. The SOFTWARE is protected by United States copyright law and international treaty provisions. You acknowledge that no title to the intellectual property in the SOFTWARE is transferred to you. You further acknowledge that title and full ownership rights to the SOFTWARE will remain the exclusive property of Quinn-Curtis, Inc. and you will not acquire any rights to the SOFTWARE except as expressly set forth in this license. You agree that any copies of the SOFTWARE will contain the same proprietary notices which appear on and in the SOFTWARE.

7. EXPORT RESTRICTIONS. You agree that you will not export or re-export the SOFTWARE to any country, person, entity, or end user subject to U.S.A. export restrictions. Restricted countries currently include, but are not necessarily limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria. You warrant and represent that neither the U.S.A. Bureau of Export Administration nor any other federal agency has suspended, revoked or denied your export privileges.

8. NO WARRANTIES. Quinn-Curtis, Inc. expressly disclaims any warranty for the SOFTWARE. THE SOFTWARE AND ANY RELATED DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OR MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE REMAINS WITH YOU.

9. LIMITATION OF LIABILITY. IN NO EVENT SHALL QUINN-CURTIS, INC. 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 SUCH DAMAGES. IN ANY EVENT, QUINN-CURTIS'S LIABILITY FOR ANY CLAIM, WHETHER IN CONTRACT, TORT, OR ANY OTHER THEORY OF LIABILITY WILL NOT EXCEED THE GREATER OF U.S. $1.00 OR LICENSE FEE PAID BY YOU.

10. U.S. GOVERNMENT RESTRICTED RIGHTS. The SOFTWARE is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer SOFTWARE clause of DFARS 252.227-7013 or subparagraphs (c)(i) and (2) of the Commercial Computer SOFTWARE- Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is: Quinn-Curtis, Inc., 18 Hearthstone Dr., Medfield MA 02052 USA.

11. MISCELLANEOUS. If you acquired the SOFTWARE in the United States, this EULA is governed by the laws of the state of Massachusetts. If you acquired the SOFTWARE outside of the United States, then local laws may apply.

Should you have any questions concerning this EULA, or if you desire to contact Quinn-Curtis, Inc. for any reason, please contact Quinn-Curtis, Inc. by mail at: Quinn-Curtis, Inc., 18 Hearthstone Dr., Medfield MA 02052 USA, or by telephone at: (508)359-6639, or by electronic mail at: support@Quinn-Curtis.com

Table of Contents

Real-Time Graphics Tools for Java

1. Introduction

New Features found in the 3.0 version of QCRTGraph

Revision 3.0 is all about the QCSPCChart software. It was rewritten to utilize the Event-Based charting added to QCChart2D a couple of years ago. No new features have been added to the QCRTGraph software. You can read about what's new in Revision 3 of QCSPCChart in the QCSPCChart manual, if you have that product, or on our website at www.quinn-curtis.com. The revision change is just to QCSPCChart, QCRTGraph and QCChart2D in sync.

New Features found in the 2.3 version of QCRTGraph

All of the new features found in the 2.3 version of QCRTGraph were added to QCChart2D - primarily a new collection of event based charting classes. The real-time scrolling routines (RTScrollFrame) will automatically work with the new event-base coordinate systems.

Event-Based Charting

A new set of classes have been added in support of new, event-based plotting system. In event-based plotting, the coordinate system is scaled to the number of event objects. Each event object represents an x-value, and one or more y-values. The x-value can be time based, or numeric based, while the y-values are numeric based. Since an event object can represent one or more y-values for a single x-value, it can be used as the source for simple plot types (simple line plot, simple bar plot, simple scatter plot, simple line marker plot) and group plot types (open-high-low-close plots, candlestick plots, group bars, stacked bars, etc.). Event objects can also store custom data tooltips, and x-axis strings. The most common use for event-based plotting will be for displaying time-based data which is discontinuous: financial markets data for example. In financial markets, the number trading hours in a day may change, and the actual trading days. Weekends, holidays, and unused portions of the day can be excluded from the plot scale, producing continuous plots of discontinuous data. The following classes have been added to the software in support of event-based charting.

- **ChartEvent** - A ChartEvent object stores the position value, the time stamp, y-values, and custom strings associated with the event.
- **EventArray** - A utility array class used to store ChartEvent objects
- **EventAutoScale** – An auto-scale class used by the EventCoordinates class.
- **EventAxis** - Displays an axis based on an EventCoordinates scale
- **EventAxisLabels** – Displays the string labels labeling the tick marks of an EventAxis

- **EventCoordinates** – Event coordinates define a coordinate system based on the the attached Event datasets
- **EventGroupDataset** – A group dataset which uses ChartEvent objects as the source of the data. It is used to feed data into the group plotting routines.
- **EventScale** – An event scale class used to convert between event coordinates and device coordinates.
- **EventSimpleDataset** - A simple dataset which uses ChartEvent objects as the source of the data. It is used to feed data into the simple plotting routines.
-

New Features found in the 2.0 version of QCRTGraph

Additional Rev. 2.0 features added to the QCRTGraph software include:

- Scrolling support for elapsed time coordinate systems
- Vertical scrolling with auto-scaling for numeric, time/date and elapsed time coordinate system.
- A collection of "Auto" classes have been added to simplify the creation of bar indicators, meters, dials, clocks, panel meters and scrolling graphs.
- A **RTProcessVarViewer** class for the grid-like display of process variable historical data in a table

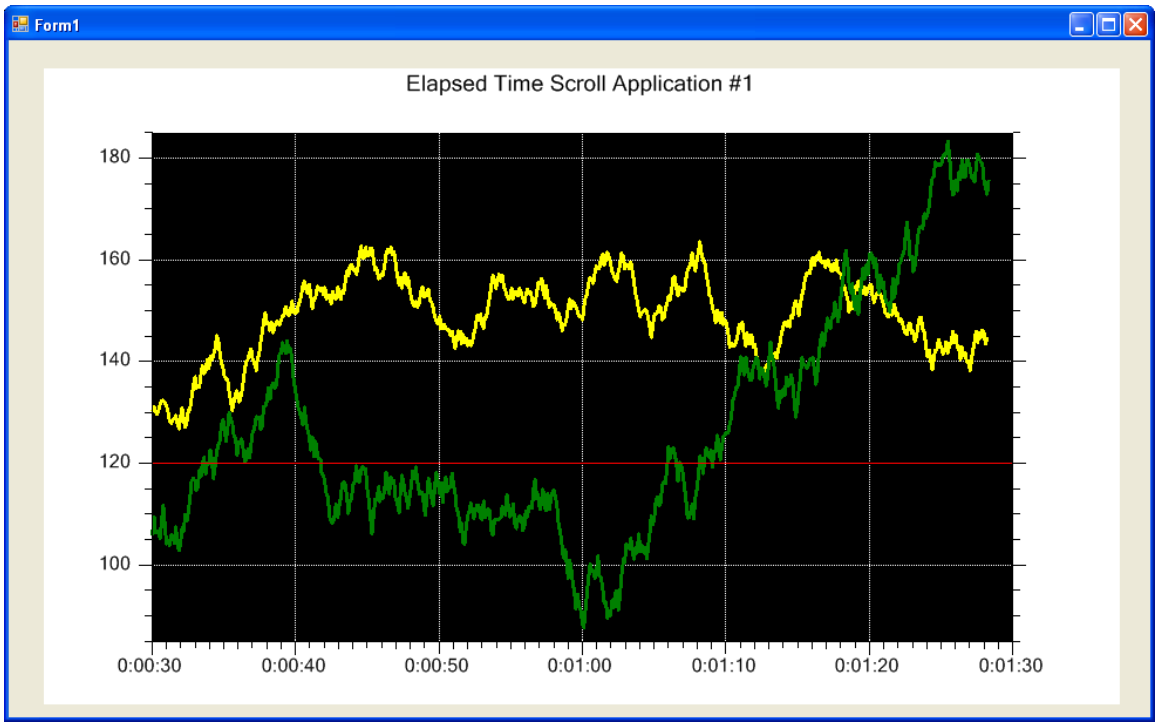
The QCRTGraph software is built on top of the QCChart2D software. Revision 2.0 has added many new features to QCChart2D. New features include:

- Five new plot types: **BoxWiskerPlot**, **FloatingStackedBarPlot**, **RingChart**, **SimpleVersaPlot** and **GroupVersaPlot**
- Elapsed time scaling to compliment the time/date scaling. Includes a set of classes specifically for elapsed time charts: **ElapsedTimeLabel**, **ElapsedTimeAutoScale**, **ElapsedTimeAxis**, **ElapsedTimeAxisLabels**, **ElapsedTimeCoordinates**, **ElapsedTimeScale**, **ElapsedTimeSimpleDataset** and **ElapsedTimeGroupDataset**.
- Vertical axis scaling for time/date and elapsed time
- A **DatasetViewer** class for the grid-like display of dataset information in a table.
- A **MagniView** class: a new way to zoom data
- A **CoordinateMove** class – used to pan the coordinate system, left, right, up, down.
- Zoom stack processing is now internal to the ChartZoom class

Refer to the QCChart2D manual for information specific to these new features.

Elapsed Time Scrolling

The **TimeCoordinates** class proved less than optimal for the display simple elapsed time scales. The software now supports elapsed time scales with the addition of **ElapsedTimeCoordinates**, **ElapsedTimeScale**, **ElapsedTimeAutoScale**, **ElapsedTimeAxis**, **ElapasedTimeLabel**, and **ElapsedTimeAxisLabels** classes. For example, you can now have a scale with a 12-hour range of (00:00:00 to 12:00:00), without an explicit calendar date associated with it. Either the x- or y-dimension can be scaled as elapsed time.

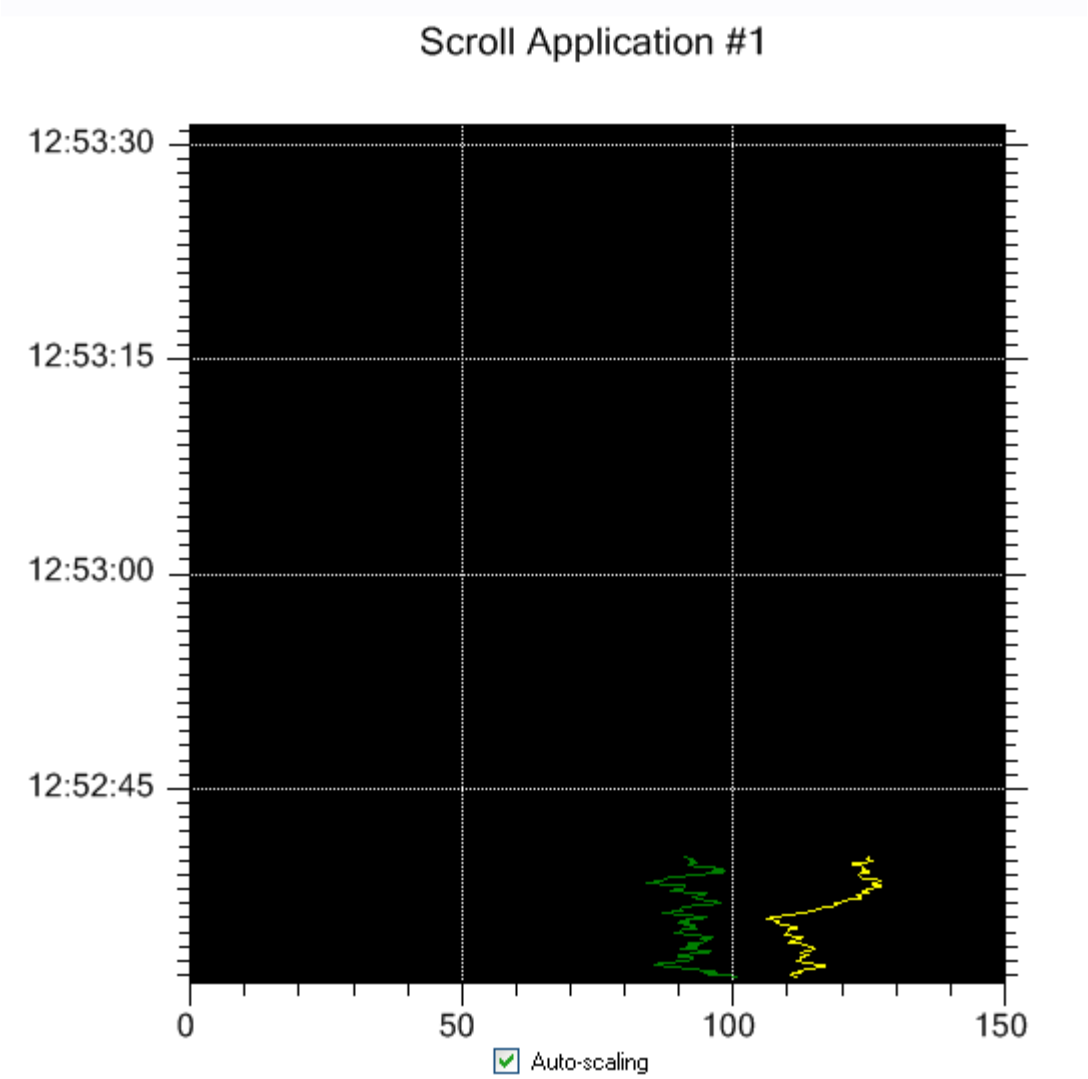


Horizontal scrolling of an elapsed time chart

Vertical Scrolling for Time/Date, Numeric and Elapsed Time Scales.

A new class, **RTVerticalScrollFrame**, manages scrolling in the vertical direction. It is compatible with the **CartesianCoordinates**, **TimeCoordinates** and the new **ElapsedTimeCoordinates** classes.

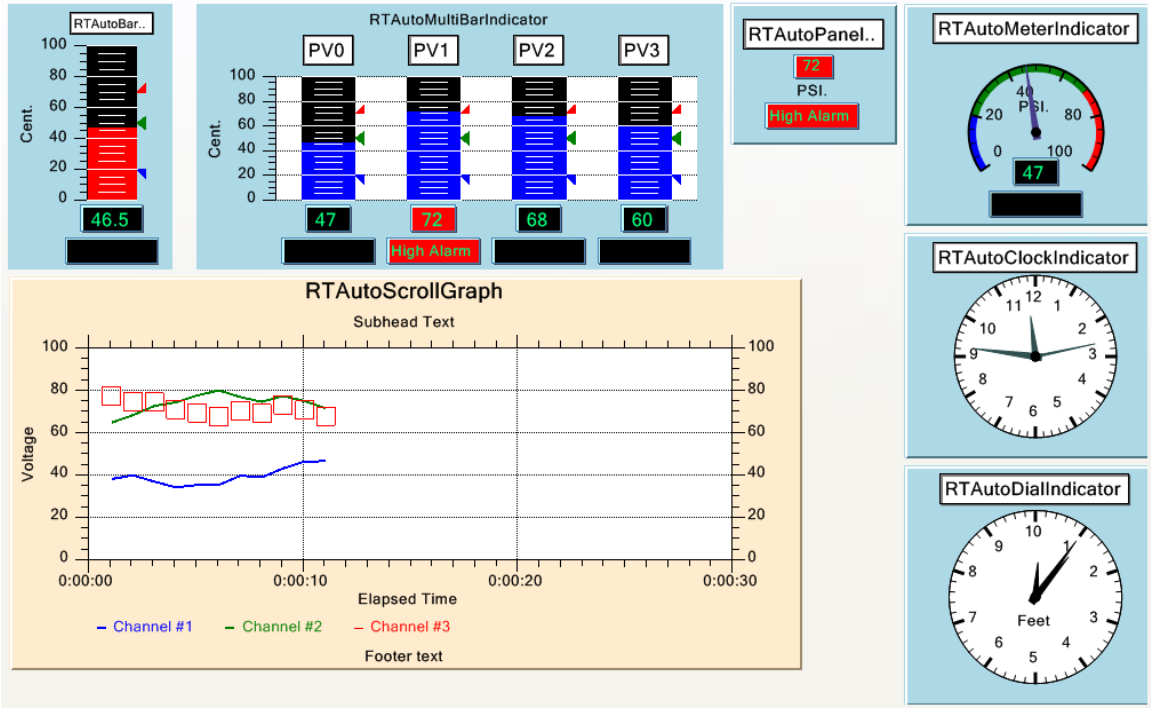
.



Time/Date Scrolling in the Vertical Dimension

New RTAuto… Indicator Classes

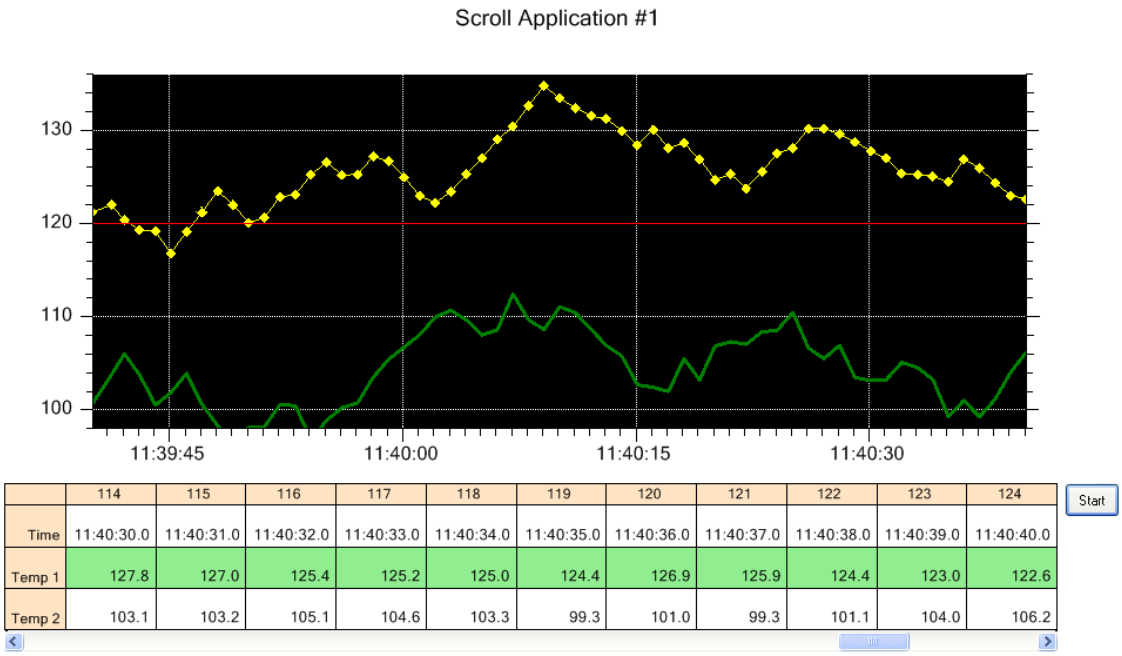
New classes have been added to simplify the creation of bar indicators, meters, dials, clocks, panel meters and scrolling graphs. These classes encapsulate all of the elements needed to create a particular real-time indicator type: coordinate system, axes, axes labels, titles, process variable, alarms, and panel meters for numeric readouts and alarm status. The auto-indicator classes are setup as a self contained ChartView derived objects, placeable on a form, and can be modified using methods and properties.



Combine the new RTAuto.. classes together on a single form.

Process Variable Data Table

Integrated data grids for viewing process variable historical data.
The new **RTProcessVarViewer** class will display **RTProcessVar** historical data using a simple grid, or table format. The viewer is derived from our QCChart2D **DatasetViewer** class. Horizontal and vertical scrolling options are supported. Numeric and time/date based formats are also supported. Row and column headers can be customized.



	114	115	116	117	118	119	120	121	122	123	124
Time	11:40:30.0	11:40:31.0	11:40:32.0	11:40:33.0	11:40:34.0	11:40:35.0	11:40:36.0	11:40:37.0	11:40:38.0	11:40:39.0	11:40:40.0
Temp 1	127.8	127.0	125.4	125.2	125.0	124.4	126.9	125.9	124.4	123.0	122.6
Temp 2	103.1	103.2	105.1	104.6	103.3	99.3	101.0	99.3	101.1	104.0	106.2

Start

View data in a table using the RTProcessVarViewer

Differences between this and the previous (1.6) version.

Elimination of the QCLicense License File.

We have eliminated the QCLicense license file from the software and with it the need to purchase additional Redistributable Licenses. Once you purchase the software, you the developer, can create application programs that use this software and redistribute the programs and our libraries royalty free. As a development tool, i.e. using this software in conjunction with a compiler, the software is still governed by a single user license and cannot be shared by multiple individuals unless additional copies, or a site license, have been purchased

Color Gradients for Solid Fill Objects

A **ChartGradient** class has been added to the software. It works in conjunction with the **ChartAttribute** class. Previously, gradients were only used for the graph and plot area backgrounds. Now, any graphical object which uses a fill color can be assigned a gradient. Unlike the .Net version of the software, where multi-color gradients are possible, gradients in Java are restricted to two colors.). Read Chapter 6 of the **QCChart2D** manual for more information about using gradients.

Tutorials

Tutorials that describe how to get started with the **Real-Time Graphics Tools for Java** charting software are found in Chapter 18 (*Compiling and Running the Example Programs*) and Chapter 19 (*Tutorial – Creating QCRTGraph for Java Applications and Applets*).

Real-Time Graphics Tools for Java Background

A large subcategory of the charting software market is concerned with the continuous or on-demand update of real-time data in a scrolling chart, gauge (bar graph), meter, annunciator or text format. Software that creates graphs of this type should make the creation and update of real-time graphs as simple and as fast as possible. The original **QCChart2D** charting product was designed to allow for the fast creation and update of custom charts using a large number of auto-scale, auto-axes, and auto-labeling routines. A good application for the **QCChart2D** software is the on-demand creation and display of historical stock market data, where the data source, time frame and scale are defined by user inputs. This is the type of charting application that you will find on Yahoo, MSN and every brokerage firm web site. A related application would involve the second by second update of real-time stock market data as it streams from a real-time data source. The software that is used for the display of historical data is seldom used to display real-time data, because its data structures are not designed for incremental updates, and its rendering routines are not fast enough to convert the data to a chart within the allowable display update interval. The **Real-Time Graphics Tools for Java** integrates the **QCChart2D** charting software with real-time data structures and specialized rendering routines. It is designed for on-the-fly rendering of the real-time data using new specialized classes for scrolling graphs, gauges (bar graphs), meters, annunciators and text. Plot objects created using the **QCChart2D** classes can be freely mixed with the new **Real-Time Graphics Tools for Java** classes. Advanced user interface features such as zooming and tool-tips can used with real-time scrolling charts.

Like the **QCChart2D** software, the **Real-Time Graphics Tools for Java** uses the graphics features found in the Java API. These include:

- Arbitrary line thickness and line styles for all lines.
- Gradients, fill patterns and color transparency for solid objects.
- Generalized geometry support used to create arbitrary shapes
- Printer and image output support
- Improved font support for a large number of fonts, using a variety of font styles, size and rotation attributes.
- Advanced matrix support for handling 2D transformation.

Real-Time Graphics Tools for Java Dependencies

The **com.quinncurtis.rtgraphjava** package is self-contained. It uses only standard classes that ship with the Java 1.2 API. In addition, all components referenced in the

software are **lightweight** components. The software uses the major Java packages listed below.

Package java.awt

The **java.awt** package is the core of Java AWT (Abstract Windowing Toolkit). It contains all of the classes that create user interfaces and draw graphics and images. The **java.awt** classes most often used in the Quinn-Curtis **chart2djava** classes are the **graphics**, **graphics2D**, **geom**, **color**, **font**, and **image**. The software also uses two graphics support packages, **java.awt.geom** and **java.awt.print,** extensively.

Class Java.awt.graphics

The **graphics** class is the abstract base class for all graphics. It contains the basic line drawing, text, area fill and color control functions needed to output a graph to an output device.

Class Java.awt.graphics2d

The **graphics2D** class extends the Graphics class to include a more sophisticated set of functions for line drawing, area filling, text placement, color control, and 2D coordinate transformations.

Class java.awt.image

This class provides routines for creating and modifying images.

Class java.awt.color

Provides a class to define colors in terms of their individual RGB (Red, Green, Blue) components.

Class java.awt.font

This class provides routines for defining and manipulating character fonts.

Package java.awt.geom

This package provides the Java 2D classes used to define and perform operations on objects related to two-dimensional geometry.

Package java.awt.print

This package provides classes and interfaces for a general printing API.

Package java.util.*

This package provides classes for tree, list and vector management, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Package java.text.*

This package provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.

Package java.io.*

This package provides classes for file i/o and serialization.

Package javax.swing.event.*

This package provides for the **MouseInputListener** events fired by Swing components.

Package javax.swing

This package provides classes of "lightweight" (all-Java language) components that, in theory if not practice, work the same on all platforms.

Directory Structure of QCRTGraph for *Java*

The **Real-Time Graphics Tools for Java** class library uses the following directory structure:

Drive:

Quinn-Curtis\ - Root directory

java\ - Quinn-Curtis Java based products directory

RedistributableFiles\ - contains the qcchart2djava.jar and qcrtgraphjava.jar files that you should use (they do not contain the javadoc files) when redistributing applications that use this library.

Docs\ - Quinn-Curtis Java related documentation directory

Lib\ - Quinn-Curtis Java related compiled libraries and components directory

com

quinn-curtis

QCRTGraph\ - Language specific code directory

Examples\ - Java examples directory

AutoGraphDemo, AnnunciatorDemo, AutoInstrumentPanel, BarApplication1, ButtonsAndTrackbars, DynBarDemo, Dynamometer, FetalMonitor, HomeAutomation, HybridCar, MeterApplication1, MeterDemo, MiniScope, Polygraph, ProcessMonitoring, ProcessVarDataTables, RTStockDisplay, RTXYDisplay, ScrollApplication1, ScrollGraphDemo, Treadmill, VerticalScrolling, WeatherStation

There are two versions of the **QCRTGraph for Java** class library: the 30-day trial version, and the developer version. Each version has different characteristics that are summarized below:

30-Day Trial Version

The trial version of **QCRTGraph for *Java*** is downloaded as a zip file named **Trial_QCRTGraphJavaR3x.zip**. The 30-day trial version stops working 30 days after the initial download. The trial version includes a version message in the upper left corner of the graph window that cannot be removed.

Developer Version

The developer version of **QCRTGraph for *Java*** is downloaded as a zip file, name something similar to JAVRTGDEV1R3x0x353x1.zip. The developer version does not time out and you can use it to create application programs that you can distribute royalty free. You can download free updates for a period of 2-years. When you placed your order, you were e-mailed download link(s) that will download the software. Those download links will remain active for at least 2 years and should be used to download current versions of the software. After 2 years you may have to purchase an upgrade to continue to download current versions of the software

Chapter Summary

The remaining chapters of this book discuss the **Real-Time Graphics Tools for Java** package designed to run on any hardware that has a Java runtime installed on it.

Chapter 2 presents the overall class architecture of the **Real-Time Graphics Tools for Java** and summarizes all of the classes found in the software.

Chapter 3 summarizes the important **QCChart2D** classes that you must be familiar with in order to use the **Real-Time Graphics Tools for Java** software.

Chapter 4 describes the process variable and alarm classes that hold **Real-Time Graphics Tools for Java** data.

Chapter 5 describes the panel meter classes: numeric, alarm, string and time/date panel meters**.**

Chapter 6 describes the single channel bar indicator classes, including segmented, custom, and pointer bar subtypes**.**

Chapter 7 describes the multi-channel bar indicator classes, including segmented, custom, and pointer bar subtypes**.**

Chapters 8 describe the meter setup classes: meter coordinates, meter axes, and meter axis labels.

Chapter 9 describes the meter indicator classes including classes for meter needles, arc, segmented arc, and symbol indicators.

Chapter 10 how the meter indicator classes are used to create dials and clocks..

Chapter 11 describes the annunciator classes.

Chapter 12 describes the scroll frame class and the implementation of scrolling plots based on the **QCChart2D SimpleLinePlot**, **SimpleBarPlot**, **SimpleScatterPlot** and **SimpleLineMarkerPlot** classes using the: **RTSimpleSingleValuePlot** class.

Chapter 13 describes the **RTGroupMultiValuePlot** class and the implementation of scrolling plots based on the **QCChart2D GroupPlot**.

Chapter 14 describes custom control classes: buttons, and track bars.

Chapter 15 describes the PID control class.

Chapter 16 describes tricks and techniques for zooming of real-time data.

Chapter 17 describes miscellaneous classes for drawing shapes and creating rectangular and circular backdrops for graphs and controls.

Chapter 18 describes the **RTProcessVarViewer** class – a data table used to display historical data collected by the RTProcessVar class.

Chapter 19 describes the new **RTAutoIndicator** classes (**RTAutoBarIndicator**, **RTAutoMultiBarIndicator**, **RTAutoMeterIndicator**, **RTAutoClockIndicator**, **RTAutoDialIndicator**, **RTAutoScrollGraph**, **RTAutoPanelMeterIndicator**) These classes simplify the creation of bar indicators, meters, dials, clocks, panel meters and scrolling graphs.

Chapter 20 is a tutorial that describes how to use **QCRTGraph** to create and run Java applications and applets

Chapter 21 is a collection of Frequently Asked Questions about **QCRTGraph for Java**.

2. Class Architecture of the Real-Time Graphics Tools for Java Class Library

Major Design Considerations

This chapter presents an overview of the **Real-Time Graphics Tools for Java** class architecture. It discusses the major design considerations of the architecture:

Based on the **QCChart2D** charting architecture, it has the same design considerations listed in that software. These are:

- It is based on the Java Graphics2D drawing API model.
- New charting objects can be added to the library without modifying the source of the base classes.
- There are no limits regarding the number of data points in a plot, the number of plots in graph, the number of axes in a graph, the number of coordinate systems in a graph.
- There are no limits regarding the number of legends, arbitrary text annotations, bitmap images, geometric shapes, titles, data markers, cursors and grids in a graph.
- Users can interact with charts using classes using **MouseInputListener** interface model.

Design consideration specific to **Real-Time Graphics Tools for Java** are:

- Updates of data classes are asynchronous with rendering of graphics to the screen.
- Real-Time plot objects are derived from **QCChart2D** plot objects resulting in standardized methods for setting plot object properties.
- Any standard plot type from the **QCChart2D** software package, both simple and group plot types, can be implemented as scrolling graphs.
- There are no limits on the number of process variable channels, no limits on the number of alarm limits associated with a process variable, no limits on the number of real-time plots in a graph.
- The update of real-time objects will not interfere or overwrite other objects and will follow the z-order precedence established when the graph was created.

The chapter also summarizes the classes in the **Real-Time Graphics Tools for Java** library.

There are five primary features of the overall architecture of the **Real-Time Graphics Tools for Java** classes. These features address major shortcomings in existing charting software for use with both Java and other computer languages.

- First, **QCChart2D for Java** uses the standard Java window architecture. Charts are placed in a **ChartView** window that derives from the **JPanel** class. Position one or more **ChartView** objects in Java container windows using the standard container layout managers. Mix charts with other components in the same container. Charts use the standard Java event listener interface model for handling mouse and keyboard events.

- The **Real-Time Graphics Tools for Java** software uses a new real-time update and rendering paradigm which represents a shift in the way Quinn-Curtis has always done real-time updates in past. In the past, graphs were always updated incrementally as new data arrived. This is no longer the case. Instead rendering is no longer incremental. When a graph is rendered, the entire graph is redrawn using the most current data. A special process variable class (**RTProcessVar**) is used to store new data as it is acquired. In the case of graphs that require a historical display of information, such as scrolling graphs, the process variable class also manages a **ChartDataset** object that holds historical information. Updating the process variable with new data values does NOT trigger a screen update. Because the screen update is not event driven from the update of the data, the process variable can be updated hundreds, or even thousands of times faster than the screen. The graph should be rendered to the screen, using a timer or some other event, at a frame rate of 10 updates/second or slower. The rendered graphs will always reflect the most current data, and in the case of scrolling graphs or other graphs that display time persistent data, will always display all data within the current scale limits. As processor speeds improve the screen updates should be able to approach the 30-60 frames/seconds of a CRT monitor. It will never need to be higher than that because the eye cannot track changes in the screen faster than that anyway.

- Since all real-time plot objects are derived from the **QCChart2D ChartPlot** class, the methods and properties of that class are available to set commonly used attributes such as the real-time plot object scale, line and fill colors.

- Many new real-time classes have been added to the software, implementing display objects that render process variable data in a variety of graph and text formats. These include single and multiple bar indicator classes, meter axis and meter indicator classes, panel meter classes, and annunciator classes. Rather than create a whole new set of classes that reproduce all of the **SimplePlot** and **GroupPlot** classes of the **QCChart2D** library, two special classes (**RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot**) are used to interface the **QCChart2D** plot objects to the process variable data classes. That way any **QCChart2D SimplePlot** or **GroupPlot**

object can be converted into a real-time scrolling graph without adding any code to the **Real-Time Graphics Tools for Java** library.

Real-Time Graphics Tools for Java Class Summary

The **Real-Time Graphics Tools for Java** library is a super set of the **QCChart2D** library. The classes of the **QCChart2D** library are an integral part of the software. A summary of the **QCChart2D** classes appears below.

QCChart2D Class Summary

Chart view class	The chart view class is a **JPanel** subclass that manages the graph objects placed in the graph
Data classes	There are data classes for simple xy and group data types. There are also data classes that handle **GregorianCalendar** date/time data and contour data.
Scale transform classes	The scale transform classes handle the conversion of physical coordinate values to working coordinate values for a single dimension.
Coordinate transform classes	The coordinate transform classes handle the conversion of physical coordinate values to working coordinate values for a parametric (2D) coordinate system.
Attribute class	The attribute class encapsulates the most common attributes (line color, fill color, line style, line thickness, etc.) for a chart object.
Auto-Scale classes	The coordinate transform classes use the auto-scale classes to establish the minimum and maximum values used to scale a 2D coordinate system. The axis classes also use the auto-scale classes to establish proper tick mark spacing values.
Charting object classes	The chart object classes includes all objects placeable in a chart. That includes axes, axes labels, plot objects (line plots, bar graphs, scatter plots, etc.), grids, titles, backgrounds, images and arbitrary shapes.
Mouse interaction classes	These classes, directly and indirectly derived from the Java **MouseInputListener** class, trap mouse events and permit the user to create and move data cursors, move plot objects, display tooltips and select data points in all types of graphs.

File and printer rendering These classes render the chart image to a printer, to a JPEG file, or to a Java **Image** object.

Miscellaneous utility classes Other classes use these for data storage, file I/O, and data processing.

For each of these categories see the associated description in the **QCChart2D** manual. The **Real-Time Graphics Tools for Java** classes are in addition to the ones above. They are summarized below.

Real-Time Graphics Tools for Java Class Summary

Process Variable and Alarms

Real-time data is stored in **RTProcessVar** classes. The **RTProcessVar** class is designed to represent a single process variable, complete with limit values, an unlimited number of high and low alarms, historical data storage, and descriptive strings for use in displays.

Single Value Indicators A single value indicator is a real-time display object that is attached to a single **RTProcessVar** object. This includes single channel bar indicators (which includes solid, segmented, custom and pointer bar indicators), meter indicators (which includes meter needles, meter arcs and meter symbol indicators), single channel annunciator indicators, panel meter indicators and scrolling graph plots based on a **QCChart2D SimplePlot** chart object.

Multiple Value Indicators A multiple value indicator is a real-time display object that is attached to a group of **RTProcessVar** objects. This includes multiple channel bar indicators (which includes solid, segmented, custom and pointer bar indicators), multiple channel annunciator indicators, panel meter indicators organized in a grid, and scrolling graph plots based on a **QCChart2D GroupPlot** chart object.

Alarm Indicators Alarm indicators are used to display alarm lines, symbols and fill areas for the **RTProcessVar** objects associated with the single value and multiple value indicator classes.

Panel Meter Classes The **RTPanelMeter** derived classes are special cases of the single value indicator classes that are used throughout the software to display real-time data in a text format. Panel

meters are available for numeric values, string values, time/date values and alarm values.

Meter Axis Classes Meter indicators needed new classes to support the drawing of meter axes, meter axis labels and meter alarm objects.

Form Control Classes The Java **JButton** and **JSlider** objects have been subclassed and enhanced for use in instrument panels. The **RTControlButton** class implements on/off colors and on/off text for momentary, toggle and radio button style buttons. The **RTTrackBar** class adds real-world scaling based on double values to the integer based **JSlider** class. **RTControlButton** and **RTTrackBar** objects can be group together in a grid, organizing the control objects functionally and visually.

Scroll Frame A scroll frame manages constant rescaling of coordinate systems of plot objects (**RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot** objects) that are displayed in a scrolling graph.

Auto Indicator Classes A group of classes encapsulate the real-time indicators (bars, meters, dials, clocks, panel meter, and scroll graphs) as self-contained **ChartView** derived classes, so that they can be placed individually on forms.

Miscellaneous Classes Support classes are used to display special symbols used for alarm limits in the software, special round and rectangular shapes that can be used as backdrops for groupings of chart objects and PID control.

Real-Time Graphics Tools for Java Classes

The **QCRTGraph** classes are a super set of the **QCChart2D** charting software. No attempt should be made to utilize the **QCRTGraph** classes without a good understanding of the **QCChart2D** classes. See the **QCChart2DJavaManual** PDF file for detailed information about the **QCChart2D** classes. The diagram below depicts the class hierarchy of the **Real-Time Graphics Tools for Java** library without the additional **QCChart2D** classes

Namespace com.quinncurtis.rtgraphjava

RTAlarmEventArgs
RTGroupDatasetTruncateArgs
RTDatasetTruncateArgs

java.util.EventListener;
RTAlarmEventListener
RTGroupDatasetTruncateListener
RTDatasetTruncateListener

Com.quinncurtis.chart2djava.ChartObj
RTAlarm
RTAlarmIndicator
RTMultiAlarmIndicator
RTProcessVar
RTComboProcessVar
RTPIDControl
RTRoundedRectangle2D
RTSymbol
RTTextFrame
RTGenShape
RT3DFrame
Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTNumericPanelMeter
RTAlarmPanelMeter
RTStringPanelMeter
RTTimePanelMeter
RTFormControlPanelMeter
RTAnnunciator
RTBarIndicator
RTMeterIndicator
RTMeterArcIndicator
RTMeterNeedleIndicator
RTMeterSymbolIndicator
RTSimpleSingleValuePlot
RTMultiValueIndicator
RTMultiValueAnnunciator
RTMultiBarIndicator
RTGroupMultiValuePlot
RTFormControlGrid
RTScrollFrame

RTVerticalScrollFrame

Com.quinncurtis.chart2djava.PolarCoordinates
RTMeterCoordinates
Com.quinncurtis.chart2djava.LinearAxis
RTMeterAxis
Com.quinncurtis.chart2djava.NumericAxisLabels
RTMeterAxisLabels
Com.quinncurtis.chart2djava.StringAxisLabels
RTMeterStringAxisLabels

javax.swing.JButton
RTControlButton
javax.swing.JSlider
RTControlTrackBar
Com.quinncurtis.chart2djava.ChartObj
RTFormControl

javax.swing.JPanel
ChartView
RTAutoIndicator
RTAutoBarIndicator
RTAutoMultiBarIndicator
RTAutoMeterIndicator
RTAutoClockIndicator
RTAutoDialIndicator
RTAutoScrollGraph
RTAutoPanelMeterIndicator

QCRTGraph Classes

Javax.swing.JPanel
ChartView
RTAutoIndicator
RTAutoBarIndicator
RTAutoMultiBarIndicator
RTAutoMeterIndicator
RTAutoClockIndicator

RTAutoDialIndicator
RTAutoScrollGraph
RTAutoPanelMeterIndicator

The **ChartView** class, found in the **QCChart2D** library, is the starting point of the **QCRTGraph** library, same as in the **QCChart2D** library. The **ChartView** class derives from the Java **javax.swing.JPanel** class**,** where the **JPanel** is the base class for the Swing collection of container objects that hold standard components such as menus, buttons, check boxes, etc. The **ChartView** class manages a collection of chart objects in a chart and automatically updates the chart objects when the underlying window processes a paint event. Since the **ChartView** class is a subclass of the **JPanel** class, it acts as a container for other Java components too.

The **ChartView** class is the base class for the self contained auto-indicator classes. Each real-time indicator is placed in its own **ChartView** derived window, along with all other objects typically associated the indicator (axes, labels, process variables, alarms, titles, etc.). Since **ChartView** is derived from **JPanel**, you can place as many auto-indicator classes on a form as you want.

RTAutoIndicator	Abstract base class for the other auto-indicator classes.
RTAutoBarIndicator	Bar indicator class displaying a single bar.
RTAutoMultiBarIndicator	Multi-bar indicator class displaying a multiple bars
RTAutoMeterIndicator	Meter indicator class displaying a single needle
RTAutoClockIndicator	Clock indicator displaying hours, minute, seconds.
RTAutoDialIndicator	Dial indicator displaying up to three needles as part of the dial
RTAutoScrollGraph	Scrolling graph can display an unlimited number of scroll graph traces
RTAutoPanelMeterIndicator	A simple panel meter indicator.

Process Variable and Alarms Classes

RTProcessVar
RTAlarm
RTAlarmEventArgs

Real-time data is stored in **RTProcessVar** classes. The **RTProcessVar** class is designed to represent a single process variable, complete with limit values, an unlimited number of high and low alarms, historical data storage, and descriptive strings for use in displays.

RTProcessVar	Real-time data is stored in **RTProcessVar** classes. The **RTProcessVar** class is designed to represent a single process variable, complete with limit values, an unlimited number of high and low alarms, historical data storage, and descriptive strings for use in displays.
RTAlarm	The **RTAlarm** class stores alarm information for the **RTProcessVar** class. The **RTAlarm** class specifies the type of the alarm, the alarm color, alarm text messages and alarm hysteresis value. The **RTProcessVar** classes can hold an unlimited number of **RTAlarm** objects in a Vector.
RTAlarmEventArgs	The **RTProcessVar** class can throw an alarm event based on either the current alarm state, or an alarm transition from one alarm state to another. The **RTAlarmEventArgs** class is used to pass alarm data to the event handler.

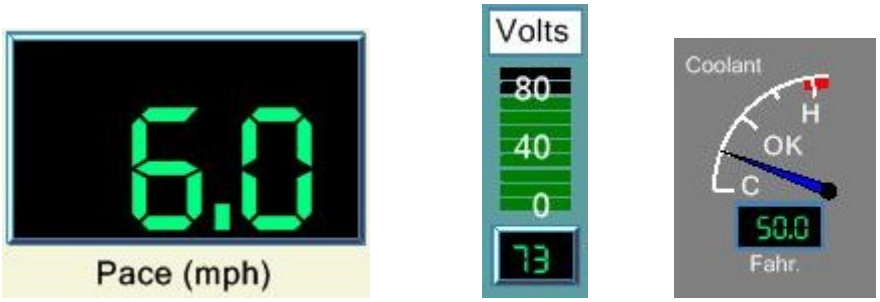
Panel Meter Classes

com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTNumericPanelMeter
RTAlarmPanelMeter
RTStringPanelMeter
RTTimePanelMeter
RTElapsedTimePanelMeter
RTFormControlPanelMeter

The **RTPanelMeter** derived classes are special cases of the single value indicator classes that are used throughout the software to display real-time data in a text format. Panel meters are available for numeric values, string values, time/date values and alarm values. All of the panel meter classes have a great many options for controlling the text font,

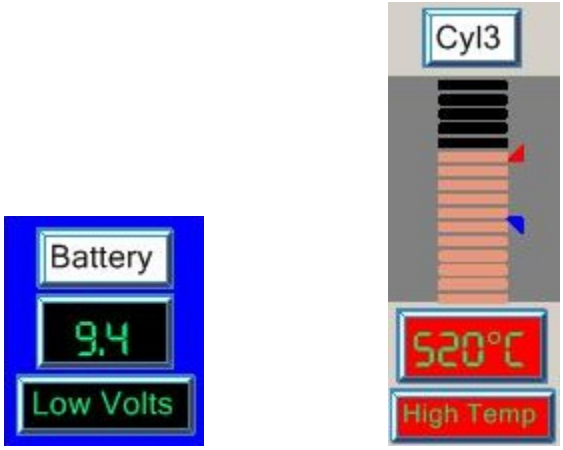
color, size, border and background of the panel meter rectangle. **RTPanelMeter** objects are used in two ways. First, they can be standalone, and once attached to an **RTProcessVar** object they can be added to a **ChartView** as any other **QCChart2D GraphObj** derived class. Second, they can be attached to most of the single channel and multiple channel indicators, such as **RTBarIndicator**, **RTMultiBarIndicator**, **RTMeterIndicator** and **RTAnnunciator** objects, where they provide text output in addition to the indicators graphical output.

RTPanelMeter The abstract base class for the panel meter types.



Numeric panel meters can be the primary display method for real-time data, or they can be used as adjuncts to other real-time indicators such as bar indicators and meters.

RTNumericPanelMeter Displays the floating point numeric value of an **RTProcessVar** object. It contains a template based on the **QCChart2D NumericLabel** class that is used to specify the font and numeric format information associated with the panel meter.



*The lowest panel meter in these examples is the **RTAlarmPanelMeter** object. Alarm properties include custom text for all alarm levels. When an alarm occurs, the foreground color of alarm text and the background color of the alarm text rectangle can be programmed to change state.*

RTAlarmPanelMeter Displays an alarm text message. It contains a template based on the **QCChart2D StringLabel** class that is used to specify the font and string format information associated with the panel meter. It bases the alarm text message on the alarm information in the associated **RTProcessVar** object.

Panel meter strings can be used to display an objects tag name, units string, and description.

RTStringPanelMeter Displays a string, either an arbitrary string, or a string based on string data in the associated **RTProcessVar** object. It is usually used to display a channels tag string and units string, but it can also be used to display longer descriptive strings. It contains a template based on the **QCChart2D**

StringLabel class that is used to specify the font and string format information associated with the panel meter.

*The **RTTimePanelMeter** can display a time/date value in any format supported by the **QCChart2D TimeLabel** format constants. You can also create custom format not directly supported.*

RTTimePanelMeter Displays the time/date value of the time stamp of the associated **RTProcessVar** object. It contains a template based on the **QCChart2D TimeLabel** class that is used to specify the font and time/date format information associated with the panel meter.

RTElapsedTimePanelMeter

Displays the elapsed time (the ChartTimeSpan value of the time stamp in milliseconds) of the associated **RTProcessVar** object. It contains a template based on the **QCChart2D ElapsedTimeLabel** class that is used to specify the font and time/date format information associated with the panel meter

RTFormControlPanelMeter

Encapsulates an **RTFormControl** object (buttons and track bars primarily, though others will also work) in a panel meter format.

Single Value Indicators

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator

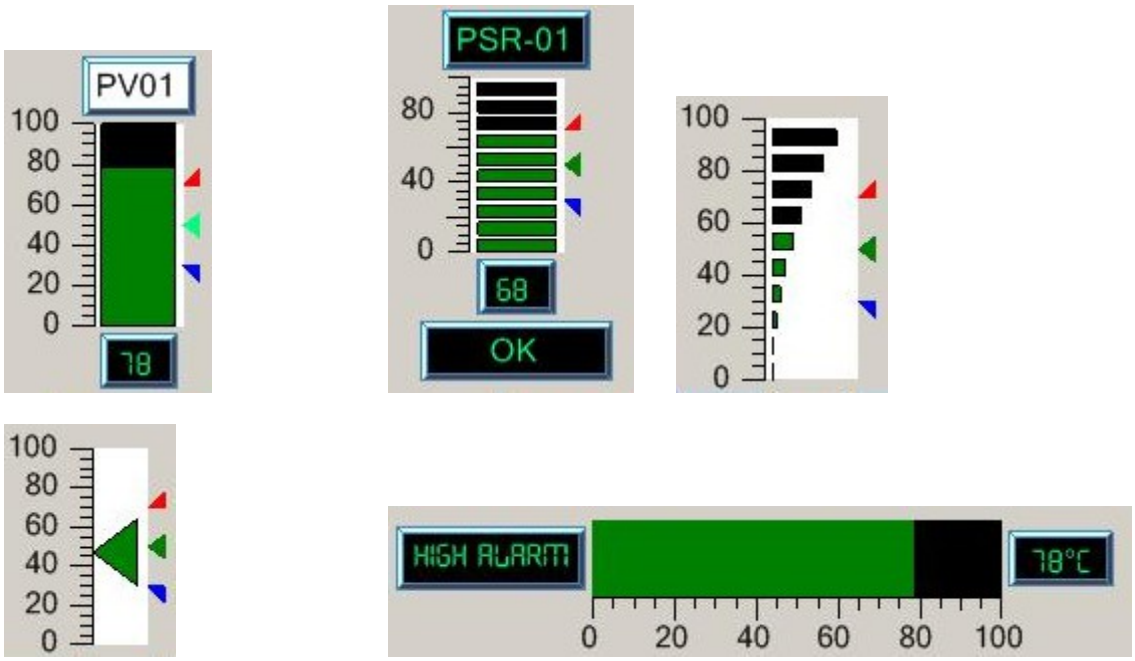
RTAnnunciator
RTBarIndicator
RTMeterIndicator
    RTMeterArcIndicator
    RTMeterNeedleIndicator
    RTMeterSymbolIndicator
RTPanelMeter
RTSimpleSingleValuePlot

Display objects derived from the **RTSingleValueIndicator** class are attached to a single **RTProcessVar** object. This includes single channel bar indicators (which includes solid, segmented, custom and pointer bar indicators), meter indicators (which includes meter needles, meter arcs and meter symbol indicators), single channel annunciator indicators, panel meter indicators and scrolling graph plots based on a **QCChart2D SimplePlot** chart object. These objects can be positioned in a chart using one of the many chart coordinate systems available for positioning, including physical coordinates (PHYS_POS), device coordinates (DEV_POS), plot normalized coordinates (NORM_PLOT_POS) and graph normalized coordinates (NORM_GRAPH_POS).

An annunciator can contain any combination string, numeric and alarm panel meters. The background color of the annunciator can change in response to an alarm event.

RTAnnunciator

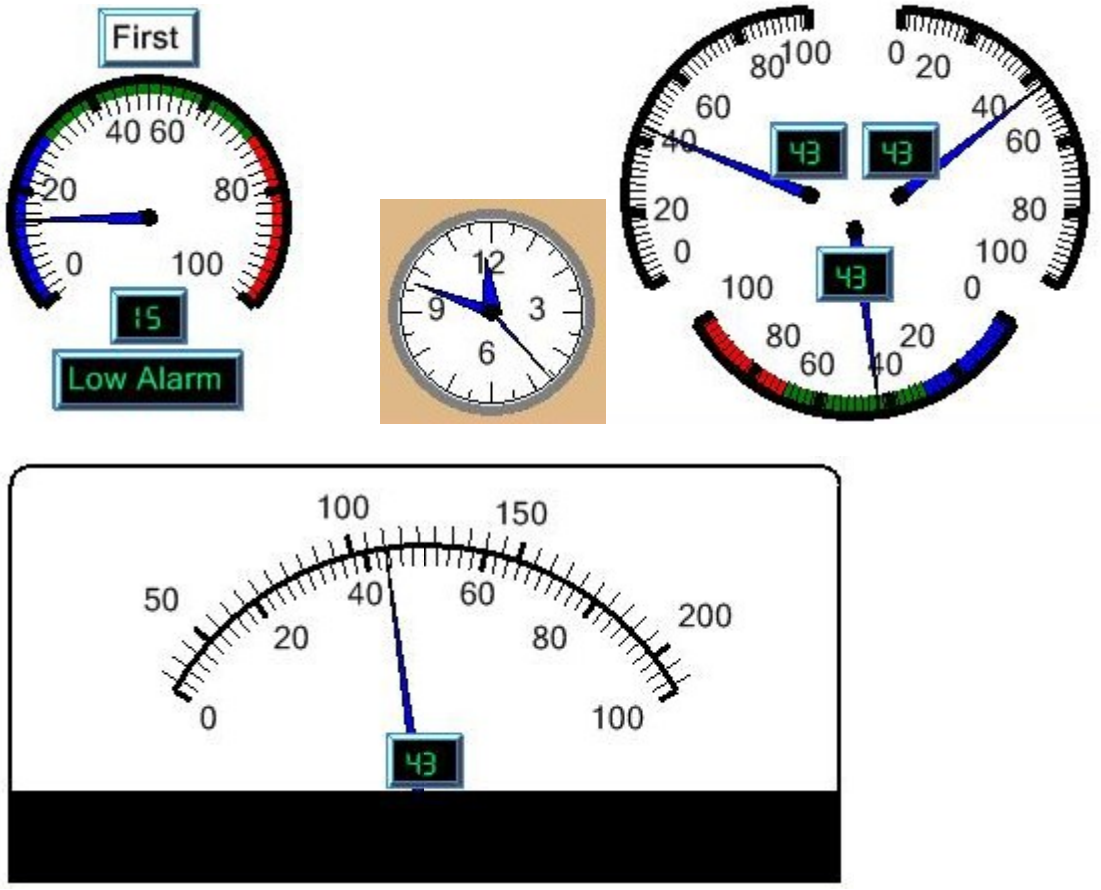
An **RTAnnunciator** is used to display the current values and alarm states of a single channel real-time data. It consists of a rectangular cell that can contain the tag name, units, current value, and alarm status message. Any of these items may be excluded. If a channel is in alarm, the background of the corresponding cell changes its color, giving a strong visual indication that an alarm has occurred.



Clockwise from top: Solid bar indicator, segmented bar indicator, custom segmented bar indicator, horizontal solid bar indicator, and pointer indicator. Only the green bars represent the bars (and pointer) represent the bar indicators. Other items also shown include axes, axis labels, panel meters, and alarm indicators.

RTBarIndicator

An **RTBarIndicator** is used to display the current value of an **RTProcessVar** using a bar changing its size. One end of each bar is always fixed at the specified base value. Bars can change their size either in vertical or horizontal direction. Sub types within the **RTBarIndicator** class support segmented bars, custom segmented bars with variable width segments, and pointer bar indicators.

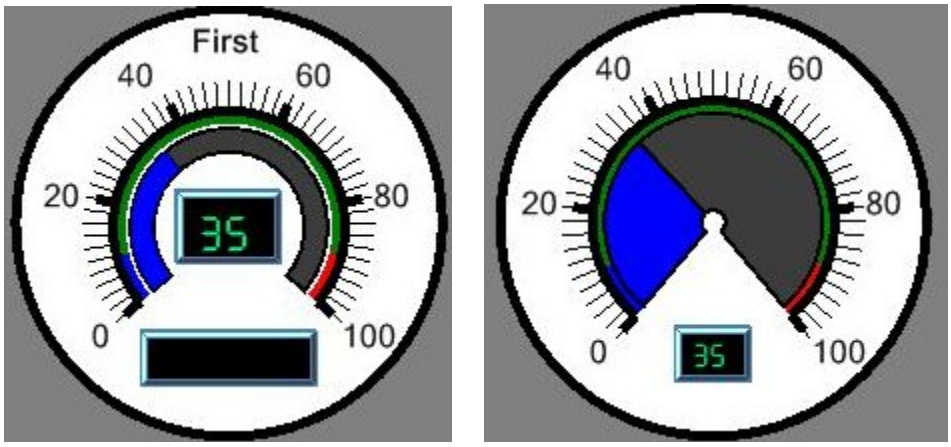


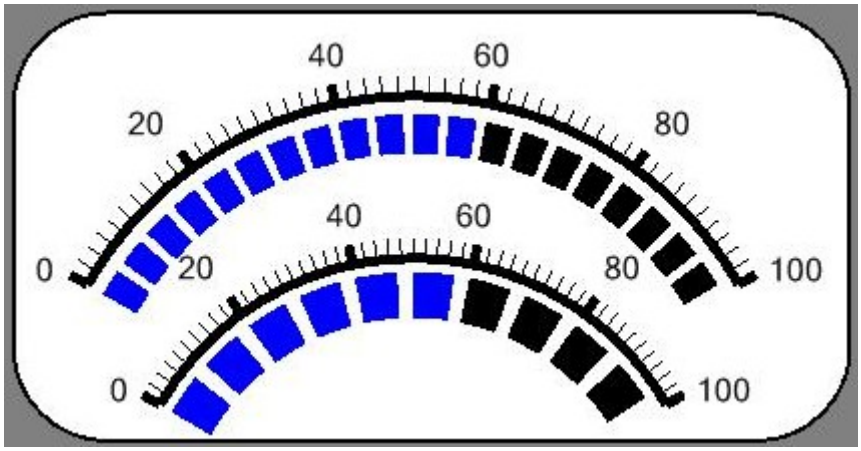
There are an infinite number of meter designs possible using a variety of meter arc ranges, meter scales, meter axes and meter indicator types

RTMeterIndicator

The **RTMeterIndicator** class is the abstract base class for all meter indicators. Familiar examples of analog meters are voltmeters, car speedometers, pressure gauges,

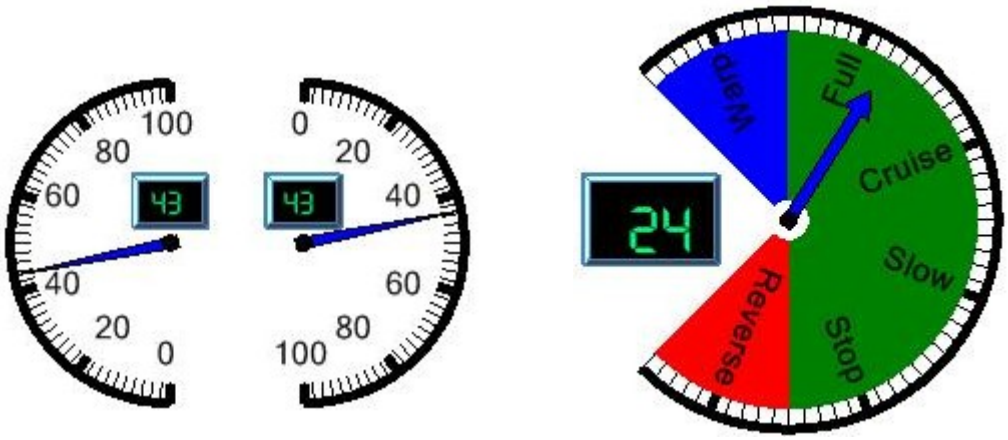
compasses and analog clock faces. Three meter types are supported: arc, symbol, and needle meters. An unlimited number of meter indicators can be added to a given meter object. **RTPanelMeter** objects can be attached to an **RTMeterIndicator** object for the display of **RTProcessVar** numeric, alarm and string data in addition to the indicator graphical display. Meter scaling, meter axes, meter axis labels and alarm objects and handle by other classes.





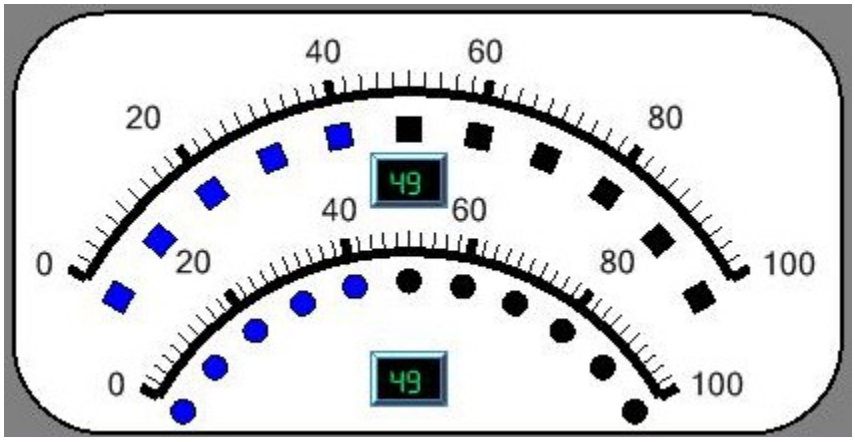
Only the blue meter arc is the arc indicator. The other elements of the meter include meter axes, meter axis labels and panel meters for the numeric, tag and alarm displays.

RTMeterArcIndicator This **RTMeterArcIndicator** class displays the current **RTProcessVar** value as an arc. Segmented meter arcs are one of the **RTMeterArcIndicator** subtypes.



Only the blue meter needles are the meter needle indicators. The other elements of the meter include meter axes, meter axis labels and panel meters for the numeric, tag and alarm displays.

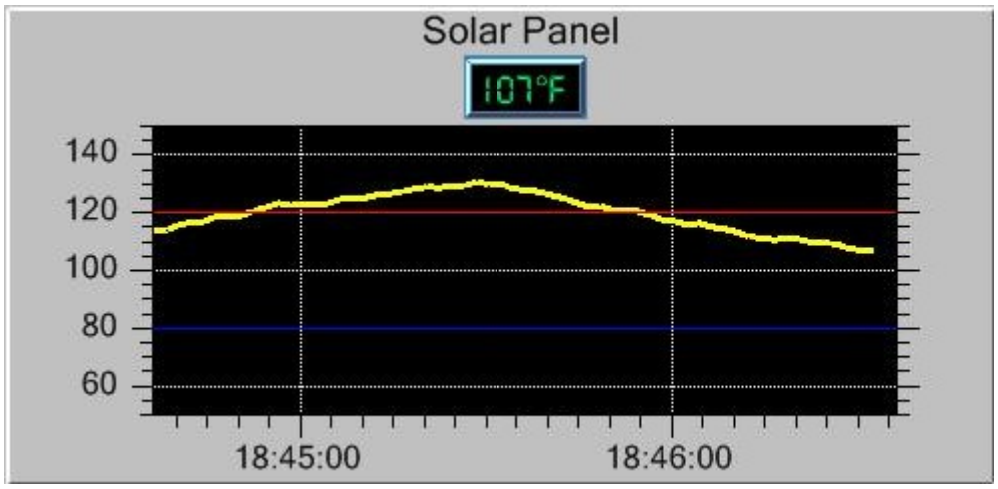
RTMeterNeedleIndicator This **RTMeterNeedleIndicator** class displays the current **RTProcessVar** value as a needle. Subtypes of the **RTMeterNeedleIndicator** are simple needles, pie wedge shaped needles and arrow needles.



Meter symbols can be any of 10 different shapes, the symbols can have any size, and the spacing between the symbols can have any value.

RTMeterSymbolIndicator This **RTMeterSymbolIndicator** class displays the current **RTProcessVar** value as a symbol moving around in the meter arc. Symbols include all of the **QCChart2D** scatter plot symbols: SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, BAR3D, CIRCLE.

RTPanelMeter The abstract base class for the panel meter types. Panel meters based objects can be added to **RTSingleValueIndicator** and **RTMultiValueIndicator** objects to enhance the graphics display with numeric, alarm and string information. The **RTNumericPanelMeter**, **RTAlarmPanelMeter**, **RTStringPanelMeter** and **RTTimePanelMeter** classes are described in the preceding section.



Any number of RTSimpleSingleValuePlot objects can be added to a scrolling graph.

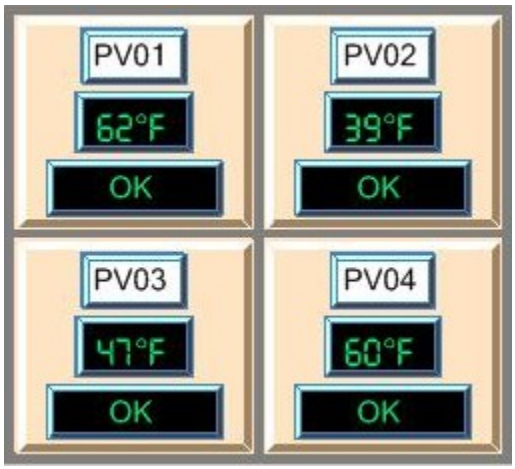
RTSimpleSingleValuePlot The **RTSimpleSingleValuePlot** plot class uses a template based on the **QCChart2D SimplePlot** class to create a real-time plot that displays **RTProcessVar** current and historical real-time data in a scrolling line, scrolling bar, or scrolling scatter plot format.

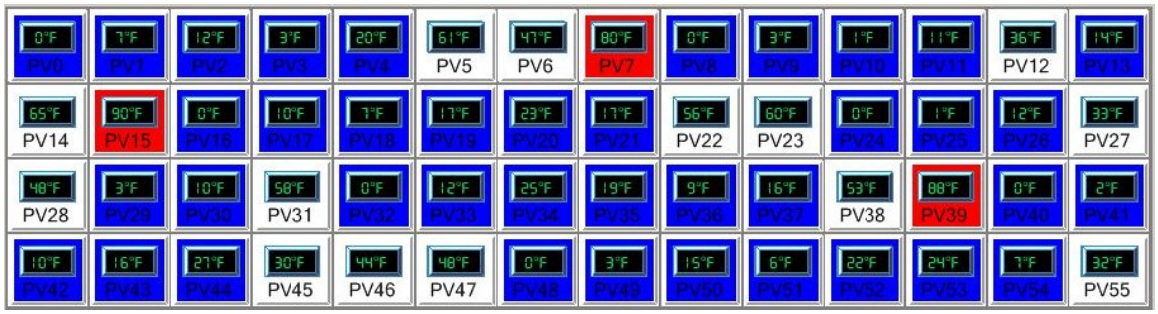
Multiple Value Indicators

Com.quinncurtis.chart2djava.ChartPlot

RTPlot
RTMultiValueIndicator
RTMultiValueAnnunciator
RTMultiBarIndicator
RTGroupMultiValuePlot
RTFormControlGrid
RTScrollFrame
RTVerticalScrollFrame

Display objects derived from the **RTMultiValueIndicator** class are attached to a collection of **RTProcessVar** objects. This includes multi-channel bar indicators (which includes solid, segmented, custom and pointer bar indicators), multi-channel annunciator indicators, and scrolling graph plots based on a **QCChart2D GroupPlot** chart object. These objects can be positioned in a chart using one of the many chart coordinate systems available for positioning, including physical coordinates (PHYS_POS), device coordinates (DEV_POS), plot normalized coordinates (NORM_PLOT_POS) and graph normalized coordinates (NORM_GRAPH_POS).

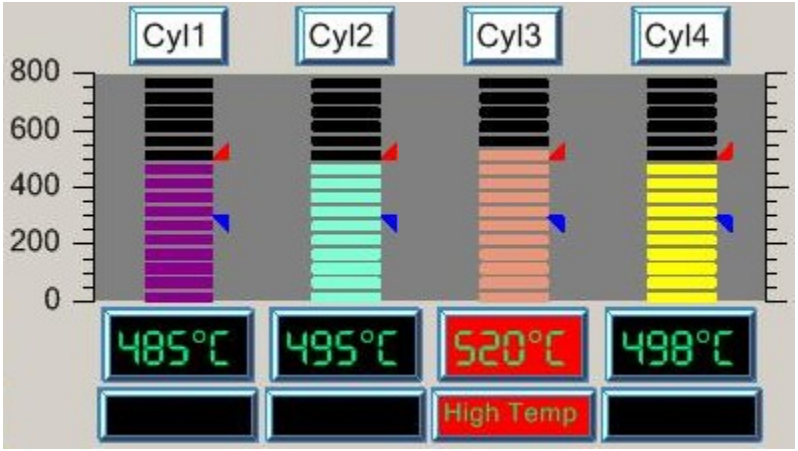




The only limit to the number of annunciator cells you can have in an ***RTMultiValueAnnunciator*** *graph is the size of the display and the readability of the text.*

RTMultiValueAnnunciator

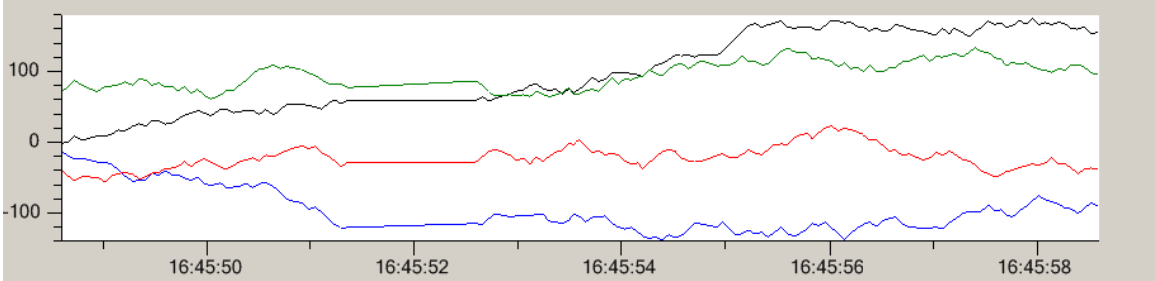
An **RTMultiValueAnnunciator** is used to display the current values and alarm states of a collection of **RTProcessVar** objects. It consists of a rectangular grid with individual channels represented by the rows and columns in of the grid. Each grid cell can contain the tag name, units, current value, and alarm status message for a single **RTProcessVar** object. Any of these items may be excluded. If a channel is in alarm, the background of the corresponding cell changes its color, giving a strong visual indication that an alarm has occurred.



*Each bar in the **RTMultiBarIndicator** can have individual colors and alarm limits.*

RTMultiBarIndicator

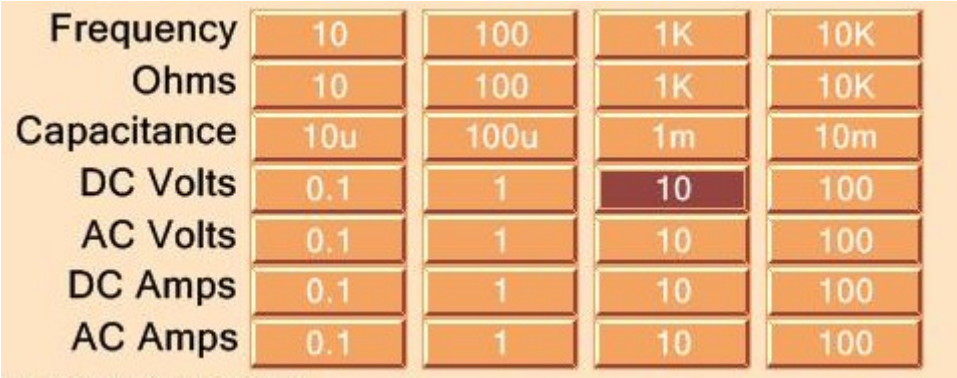
An RTMultiBarIndicator is used to display the current value of a collection of RTProcessVar objects using a group of bars changing size. The bars are always fixed at the specified base value. Bars can change their size either in vertical or horizontal direction. Sub types within the RTMultiBarIndicator class support segmented bars, custom segmented bars with variable width segments, and pointer bar indicators.

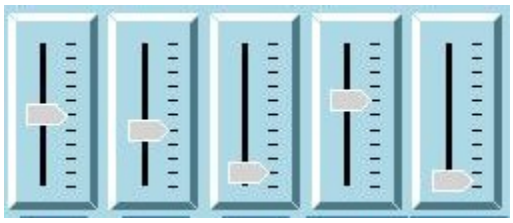


*The **RTGroupMultiValuePlot class** turns **QCChart2D GroupPlot** objects, like the **MultiLinePlot** object above, into scrolling plots.*

RTGroupMultiValuePlot

The **RTGroupMultiValuePlot** plot class uses a template based on the **QCChart2D GroupPlot** class to create a real-time plot that displays a collection of **RTProcessVar** objects as a group plot in a scrolling graph.





*The **RTFormControlGrid** class organizes **RTFormControl** objects functionally and visually.*

RTFormControlGrid

The **RTFormControlGrid** plot class organizes a group of **RTFormControl** objects (buttons and track bars primarily, though others will also work) in a grid format.

*This **RTScrollFrame** combines an **RTGroupMultiValuePlot** (the open-high-low-close plot) with two **RTSimpleSingleValuePlot** plot.*

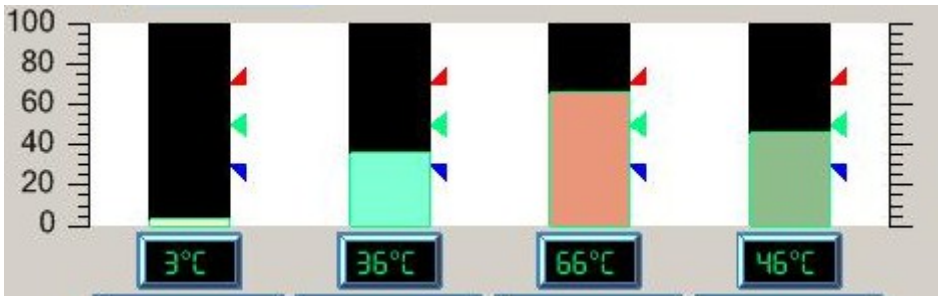
RTScrollFrame — The **RTScrollFrame** plot manages horizontal scrolling and auto-scaling for the **RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot** objects. The **RTScrollFrame** class is discussed in more detail a couple of sections down.

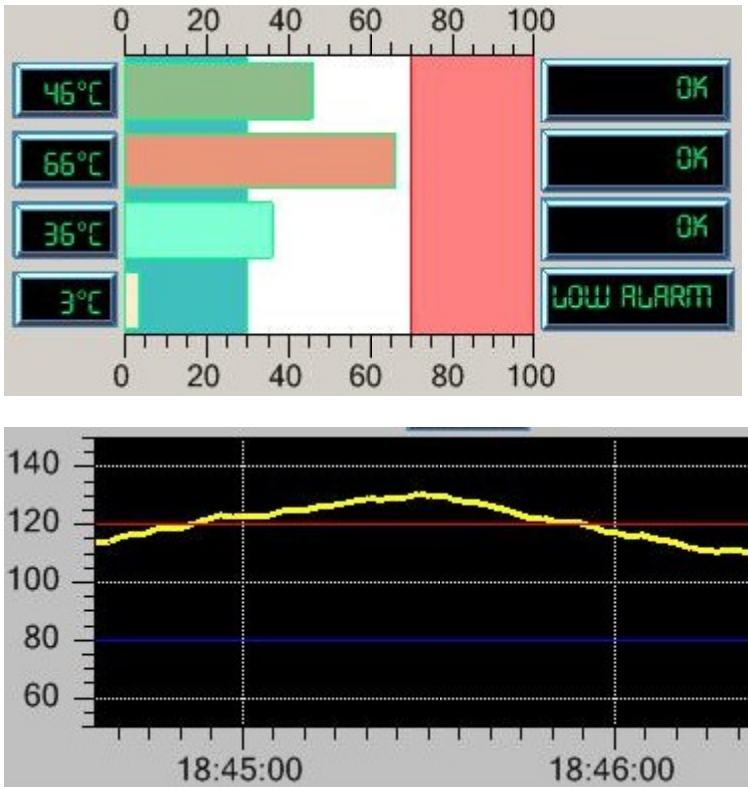
RTVerticalScrollFrame — The **RTVerticalScrollFrame** plot manages vertical scrolling and auto-scaling for the **RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot** objects. The **RTScrollFrame** class is discussed in more detail a couple of sections down.

Alarm Indicator Classes

RTAlarmIndicator
RTMultiAlarmIndicator

The alarm indicator classes are used to indicate alarms limits in displays that use a Cartesian (XY) coordinate system. The alarm indicators can have one of three forms: pointer style symbols, horizontal or vertical lines, or horizontal or vertical filled areas. These alarm indicator classes are not used in meter displays. Alarm limits for meter displays are handled by the **RTMeterAxis** class.





The alarm indicators can have one of three forms: pointer style symbols, horizontal or vertical lines, or horizontal or vertical filled areas.

RTAlarmIndicator This class is used to provide alarm limit indicators for **RTSingleValueIndicator** objects.

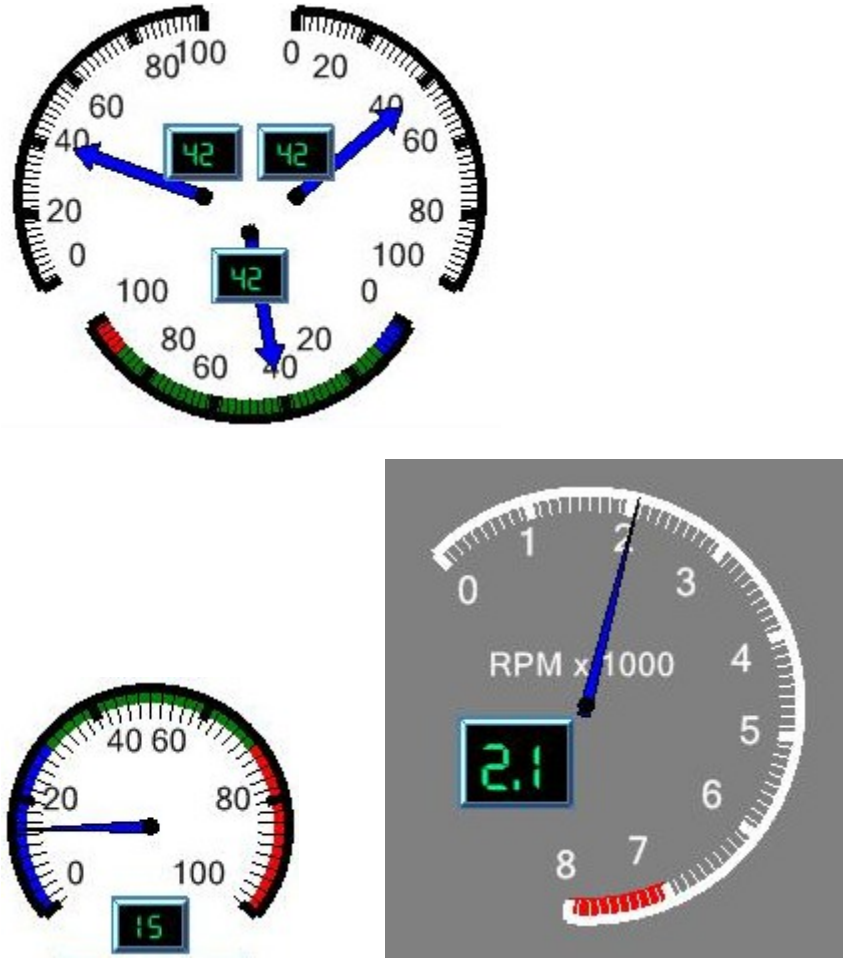
RTMultiAlarmIndicator This class is used to provide alarm limit indicators for **RTMultiValueIndicator** objects. Each indicator in a multi-indicator object can have unique alarm settings.

Meter Axis Classes

QChart2D.PolarCoordinates
RTMeterCoordinates
Com.quinncurtis.chart2djava.LinearAxis
RTMeterAxis
Com.quinncurtis.chart2djava.NumericAxisLabels
RTMeterAxisLabels
Com.quinncurtis.chart2djava.StringAxisLabels

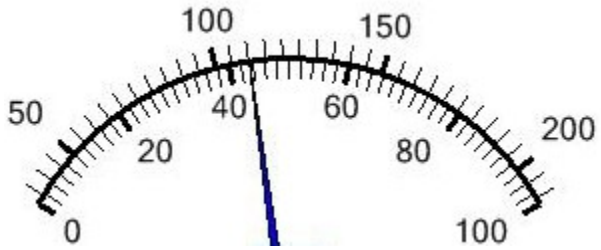
RTMeterStringAxisLabels

RTMeterCoordinates **A meter coordinate system has more properties than a simple Cartesian coordinate system, or even a polar coordinate system. Because of the variation in meter styles, a meter coordinate system sets the start and end angle of the meter arc within the 360 degree polar coordinate system. It also maps a physical coordinate system, representing the meter scale, on top of the meter arc. And the origin of the meter coordinate system can be offset in both x- and y-directions with respect to the containing plot area.**



A meter axis can have any number of alarm arcs. A meter can have multiple axes representing multiple scales

RTMeterAxis A meter axis extends for the extent of the meter arc and is centered on the origin. Major and minor tick marks are placed at evenly spaced intervals perpendicular to the meter arc. The meter axis also draws meter alarm arcs using the alarm information in the associated **RTProcessVar** object.



A useful feature of multiple meter scales is the ability to display both Fahrenheit and Centigrade scales at the same time.

RTMeterAxisLabels This class labels the major tick marks of the **RTMeterAxis** class. The class supports many predefined and user-definable formats, including numeric, exponent, percentage, business and currency formats.

Meter tick mark strings can be horizontal, parallel and perpendicular to the tick mark.

RTMeterStringAxisLabels This class labels the major tick marks of the **RTMeterAxis** class using user-defined strings

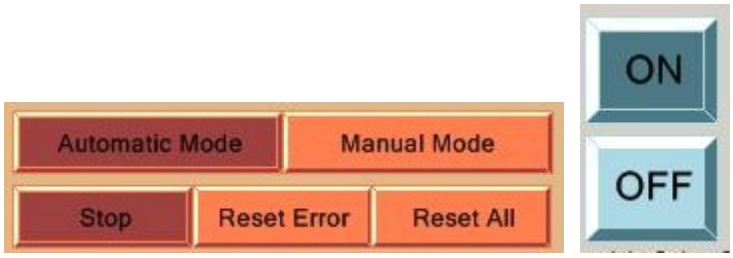
Form Control Classes

javax.swing.JButton
    RTControlButton
javax.swing.JSlider
    RTControlTrackBar
Com.quinncurtis.chart2djava.ChartObj
    RTFormControl
    RTPanelMeter
    RTFormControlPanelMeter
    RTMultiValueIndicator
    RTFormControlGrid

Real-time displays often require user interface features such as buttons and track bars. The Java Swing package includes a large number of useful controls. The **JButton, JSlider, JScrollBar** are examples of what we refer collectively as Form Controls. Sometime though the Form controls have shortcomings.. The **JScrollBar** and the **JSlider** controls have the fault that they work only with an integer range of values. The **JButton** controls are momentary and require extra programming in order to use them as toggle buttons or radio buttons..

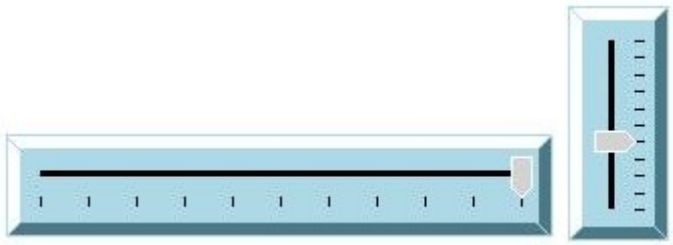
We created subclassed versions of the **JSlider** control and the **JButton** control. Our version of the **JSlider** control is **RTControlTrackBar** and adds floating point scaling for the track bar endpoints, increments, current value and tick mark frequency. Our version of the **JButton** control is **RTControlButton** adds superior On/Off button text and color control and supports momentary, toggle and radio button styles.

No matter what Form Control is used, either ours or the original Java Swing controls, it can be used in conjunction with the **RTFormControl**, **RTFormControlPanelMeter** and **RTFormControlGrid** classes.



The RTControlButton type supports momentary, toggle and radio button styles.

RTControlButton — Derived from the Java Swing **JButton** class, it adds superior On/Off button text and color control and supports momentary, toggle and radio button styles.



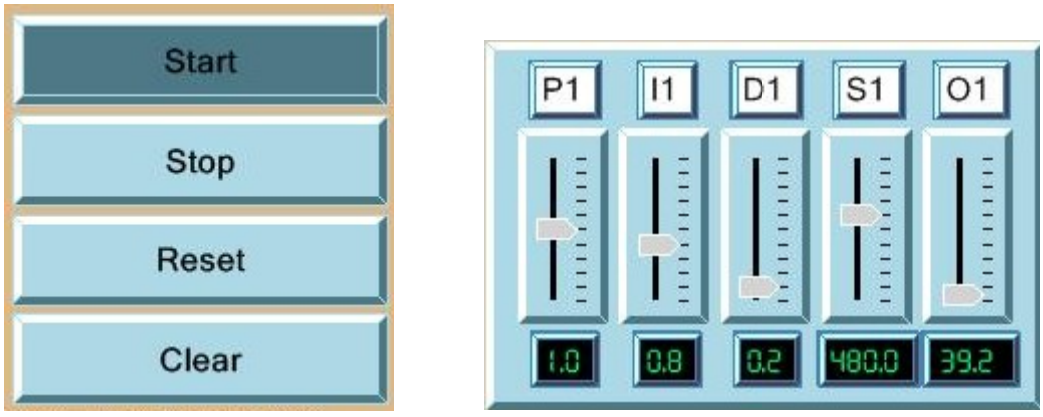
Horizontal and vertical track bars can be scaled for physical real world coordinates.

RTControlTrackBar Derived from the Java Swing **JSlider** class, it adds floating point scaling for the track bar endpoints, increments, current value and tick mark frequency.

RTFormControl The **RTFormControl** class wraps the Java Swing controls, and our **RTControlButton** and **RTControlTrackBar** controls so that they can be placed in a graph.

RTFormControlPanelMeter

This panel meter class contains encapsulates an **RTFormControl** object in a panel meter class, so that controls can be added to indicator objects.

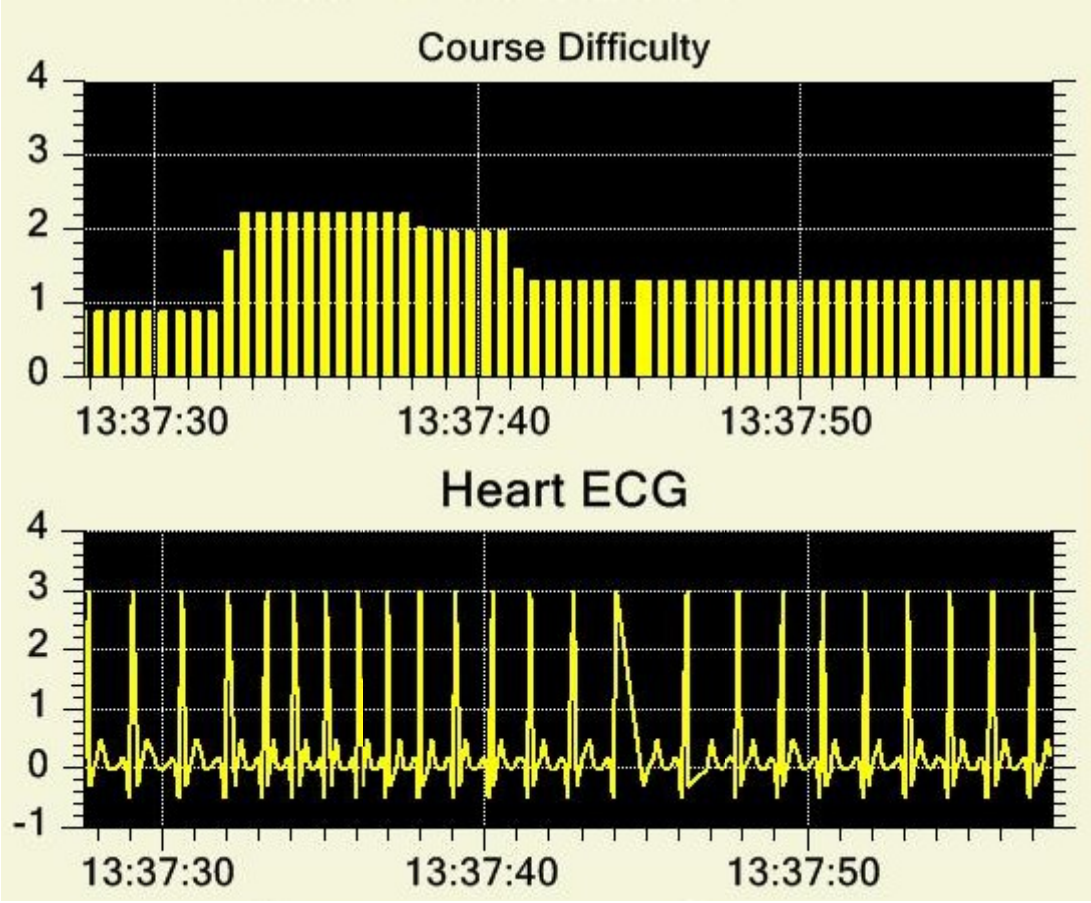


RTFormControlGrid *objects are arranged in a row x column format. Additional panel meter objects (numeric and string panel meters in the track bar example above) can be attached to the primary control grid object.*

RTFormControlGrid The **RTFormControlGrid** organizes a collection of **RTFormControl** objects functionally and visually in a grid format. An **RTControlButton** must be added to an **RTFormControlGrid** before the radio button processes of the **RTControlButton** will work.

Scroll Frame Class

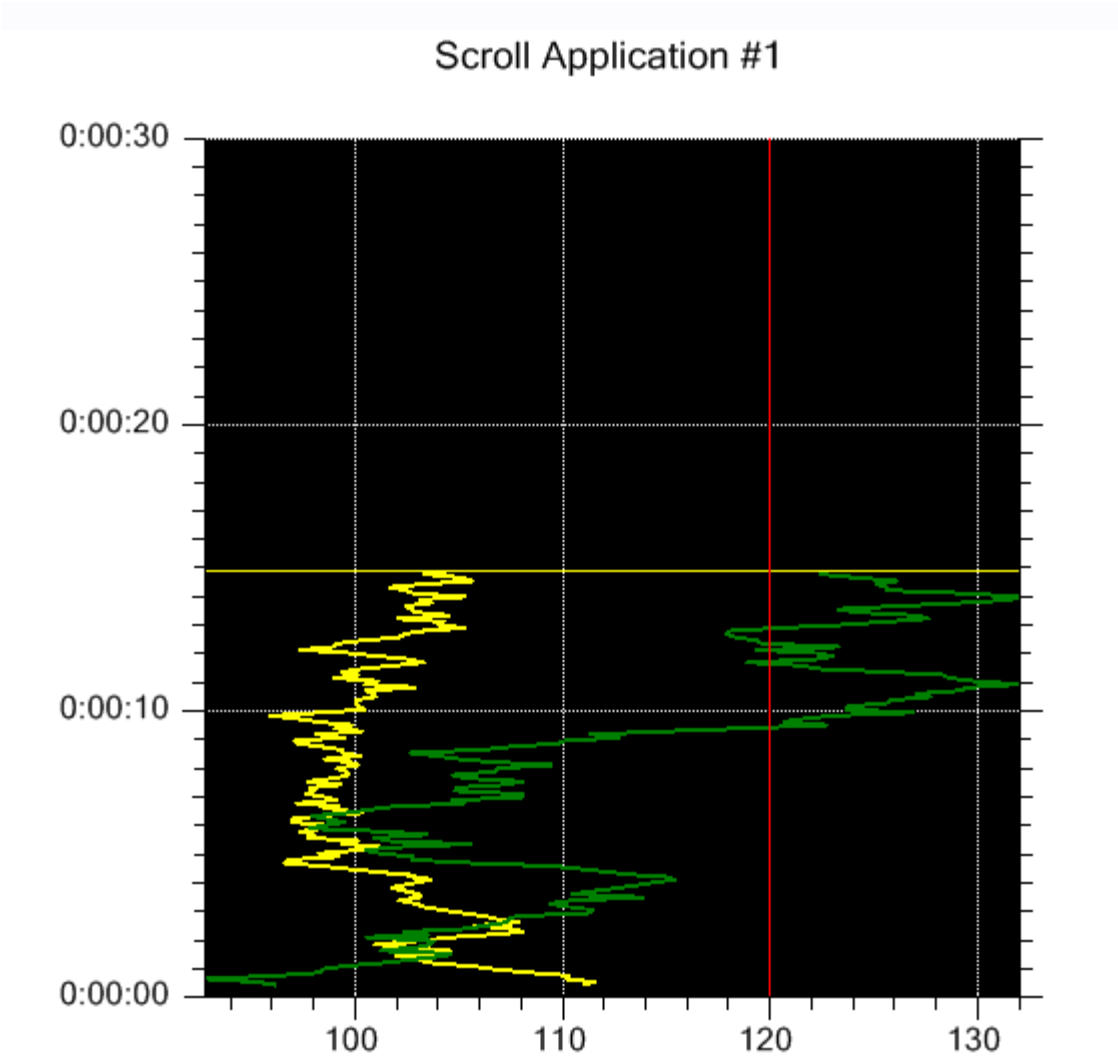
Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTMultiValueIndicator
RTScrollFrame
RTVerticalScrollFrame



A display can have multiple scroll frames. The frames can be in separate plots and update in a synchronized fashion, or they can overlap the same plotting area.

RTScrollFrame

The scrolling algorithm used in this software is different that in earlier Quinn-Curtis real-time graphics products. Scrolling plots are no longer updated incrementally whenever the underlying data is updated. Instead, the underlying **RTProcessVar** data objects are updated as fast as you want. Scrolling graphs (all graphs for that matter) are only updated with the ChartView.updateDraw() method is called. What makes scrolling graphs appear to scroll is the scroll frame (**RTScrollFrame**). When a scroll frame is updated as a result of the ChartView.updateDraw() event, it analyze the **RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot** objects that have been attached to it and creates a coordinate system that matches the current and historical data associated with the plot objects. The plot objects in the scroll frame are drawn into this coordinate system. As data progresses forward in time the coordinate system is constantly being rescaled to include the most recent time values as part of the x-coordinate system. You can control whether or not the starting point of the scroll frame coordinate system remains fixed, whether it advances in sync with the constantly changing end of the scroll frame. Other options allow the y-scale to be constantly rescaled to reflect the current dynamic range of the y-values in the scroll frame.



An example of a vertical elapsed time scroll frame.

RTVerticalScrollFrame

The **RTScrollFrame** has the limitation that it can only manage horizontal scrolling. The **RTVerticalScrollFrame** is much the same as **RTScrollFrame**, only it manages scrolling in the vertical direction. The **RTVerticalScrollFrame vertically scrolls data with a numeric, time/date or elapsed time, time stamp.**

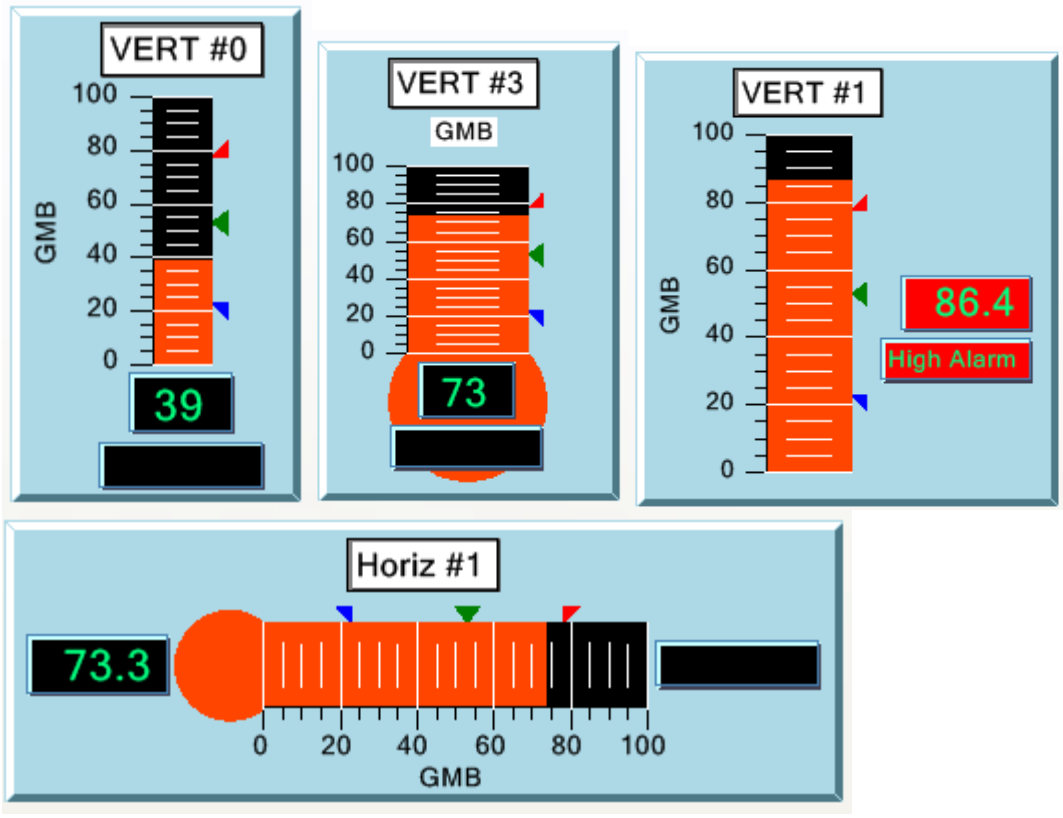
Auto Indicator Classes

JPanel

- **ChartView**
- **RTAutoIndicator**
  - **RTAutoBarIndicator**
  - **RTAutoMultiBarIndicator**
  - **RTAutoMeterIndicator**

RTAutoDialIndicator
RTAutoClockIndicator
RTAutoPanelMeterInicator

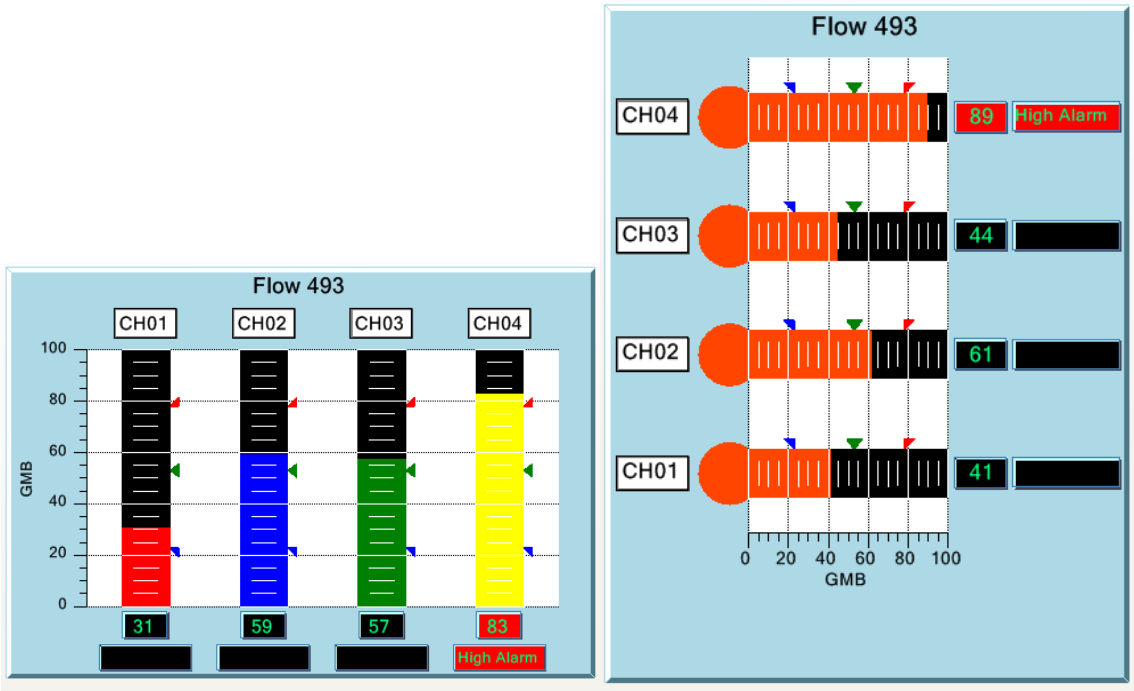
The **ChartView** class is the base class for the self contained auto-indicator classes. Each real-time indicator is placed in its own **ChartView** derived window, along with all other objects typically associated the indicator (axes, labels, process variables, alarms, titles, etc.). Since **ChartView** is derived from **JPanel**, you can place as many auto-indicator classes on a form as you want.



The RTAutoBarIndicator has many different format options for self-contained, single channel, bar indicators.

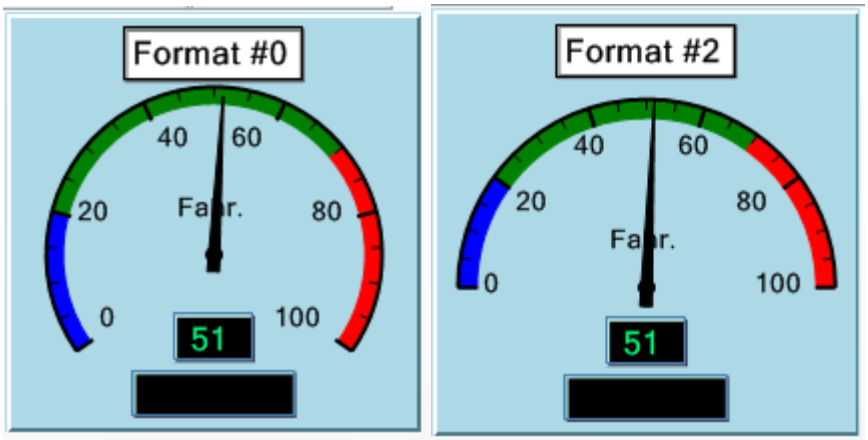
RTAutoBarIndicator

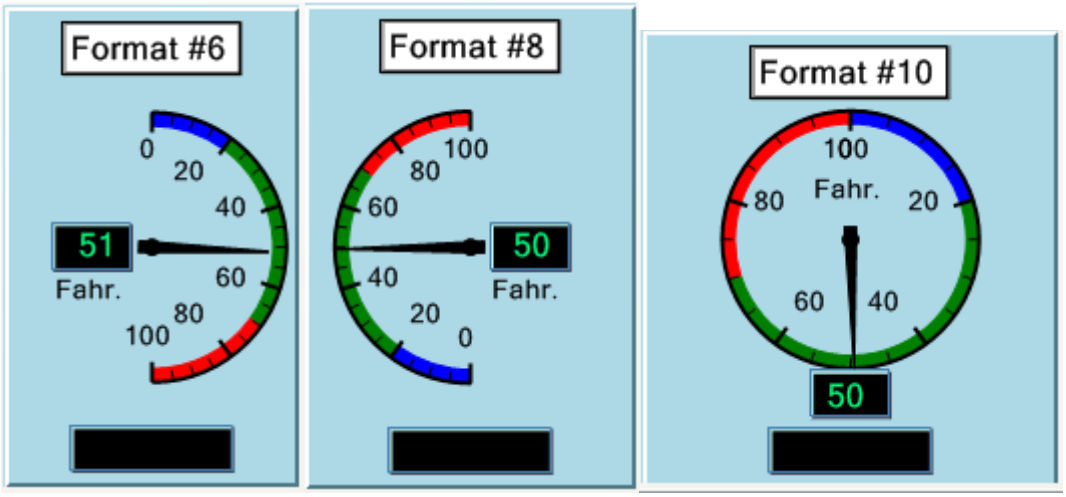
An **RTAutoBarIndicator** is a self contained control derived from the **ChartView** JPanel control. It is used to display the current value of a single channel of real-time data, and includes as options a numeric readout, alarm status readout, title, units, alarm indicators, and a bar end bulb. The indicator can be horizontal or vertical, with four different format options for each.



The RTAutoMultiBarIndicator displays multiple channels of real-time data in a single chart.

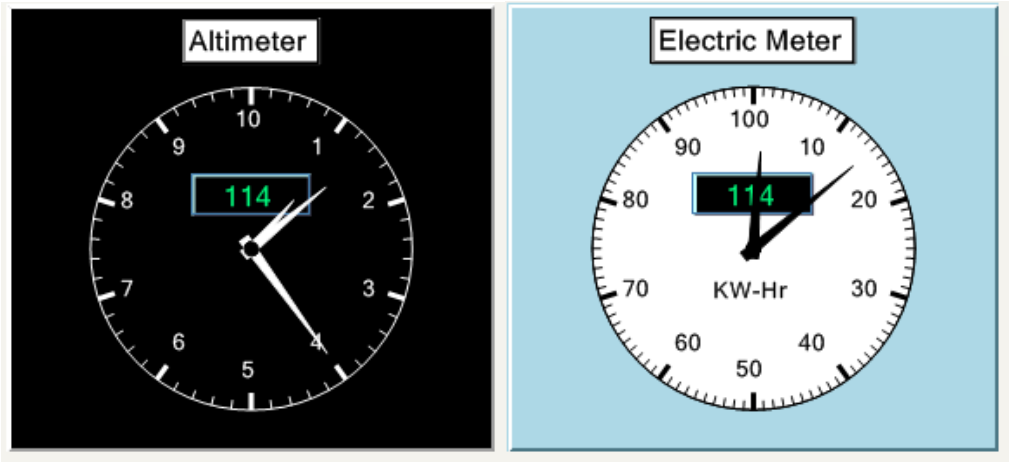
RTAutoMultiBarIndicator An **RTAutoMultiBarIndicator** is a self contained control derived from the **ChartView** JPanel control. It is used to display the current values of multiple channels of real-time data, and includes as options numeric readouts, alarm status readouts, channel names, title, units, alarm indicators, and bar end bulbs. The indicator can be horizontal or vertical with two format options for each.





The RTAutoMeterIndicator has many different format options for self-contained, single channel, meter indicators.

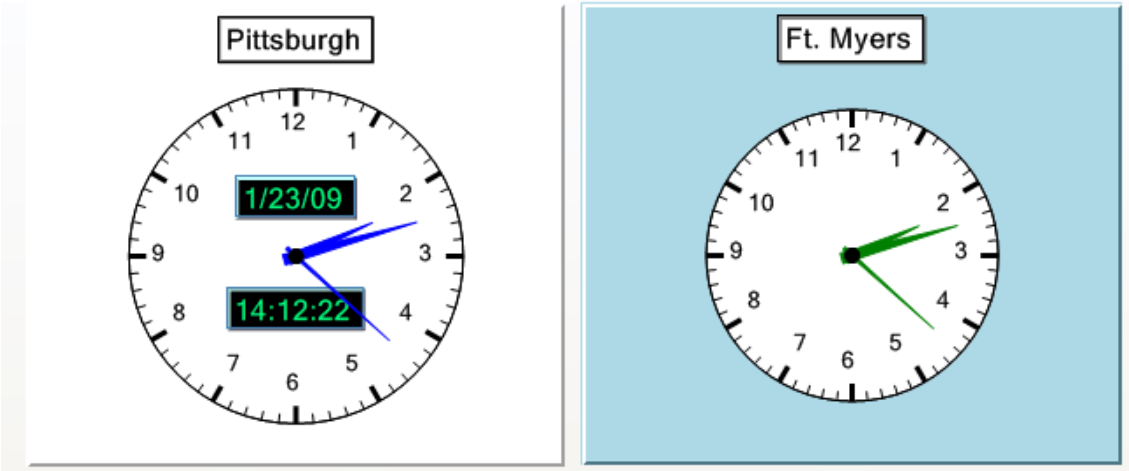
RTAutoMeterIndicator An **RTAutoMeterIndicator** is a self contained control derived from the **ChartView** JPanel control. It is used to display the current value of a single channel of real-time data, and includes as options a numeric readout, alarm status readout, title, units, and alarm arcs. There are twelve different auto meter formats.



The RTAutoDialIndicator is able to take a single numeric value and divide it into multiple needle values.

RTAutoDialIndicator An **RTAutoDialIndicator** is a self contained control derived from the **ChartView** JPanel control. It is used to display the values of up to three channels of real-time data,

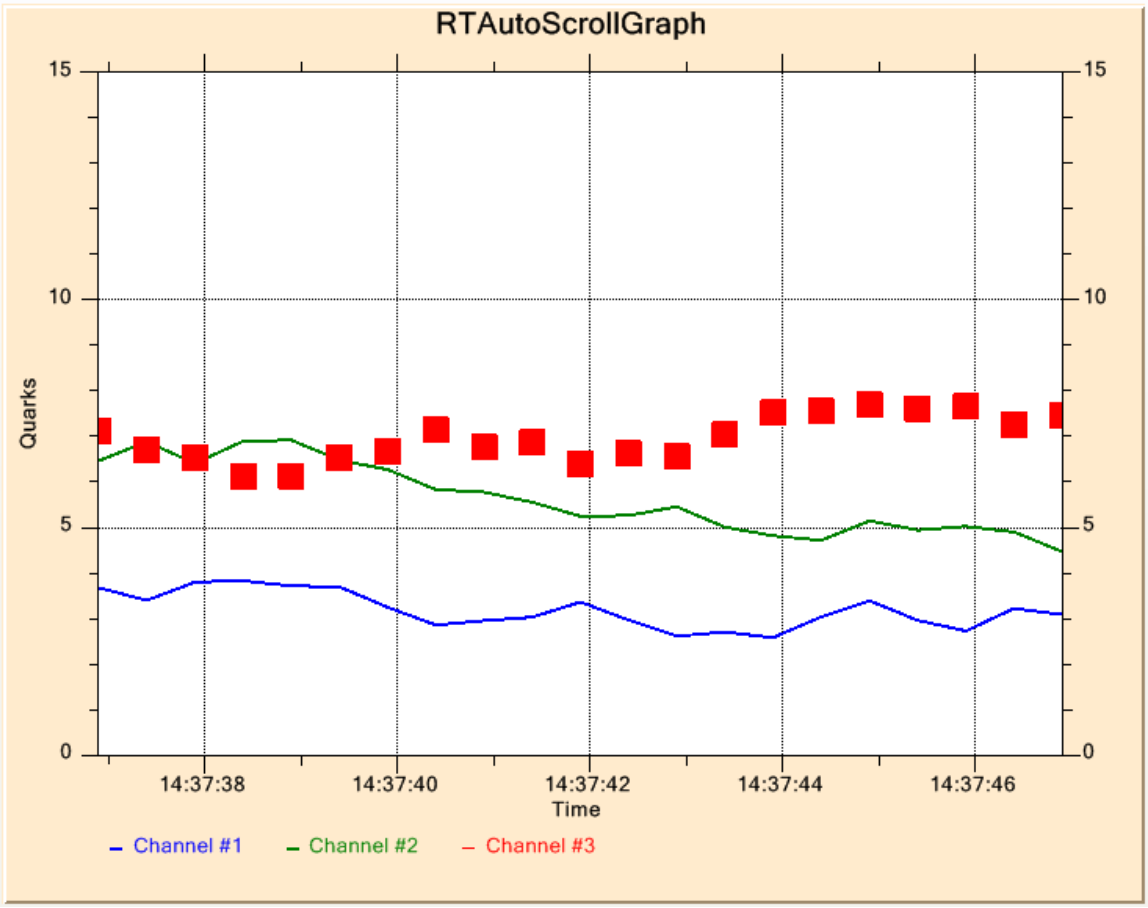
and includes as options a numeric readout, alarm status readout, title, and units.



The RTAutoClockIndicator can display the time and date in numeric format.

RTAutoClockIndicator

An **RTAutoClockIndicator** is a self contained JPanel control derived from the **ChartView** control. It is used to display the values of up to three channels of real-time data, and includes as options a numeric readout, alarm status readout, title, and units.



RTAutoScrollGraph

An **RTAutoScrollGraph** is a self contained JPanel control derived from the **ChartView** control. It is used to display real-time data in a variety of plot formats: line plots, bar plots, scatter plots and line marker plots. Options include a horizontal or vertical display, automatic legend, a main title, and axis titles.

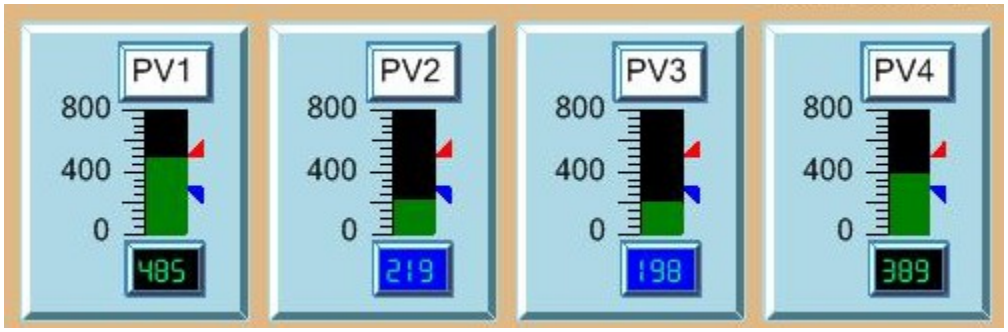
Miscellaneous Classes

Support classes are used to display special symbols used for alarm limits in the software, special round and rectangular shapes that can be used as backdrops for groupings of chart objects and PID control.

Miscellaneous Classes

Com.quinncurtis.chart2djava.GraphObj
RT3DFrame
Com.quinncurtis.chart2djava.GraphObj
RTGenShape
Com.quinncurtis.chart2djava.ChartObj
RTPIDControl

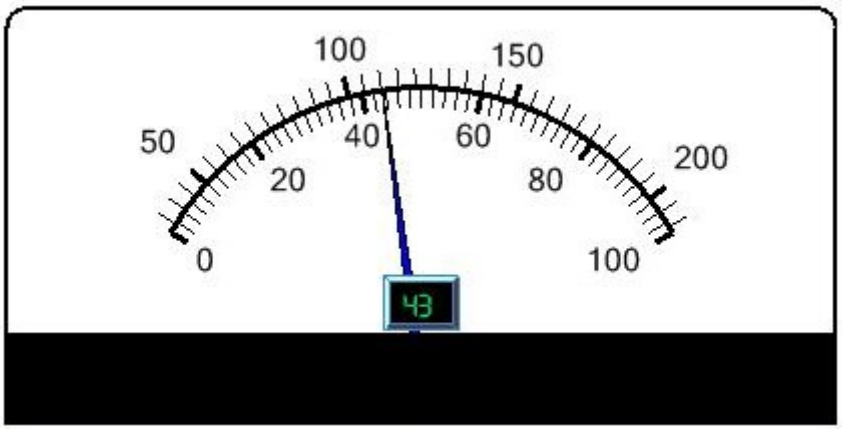
Com.quinncurtis.chart2djava.GraphObj
RTSymbol
Com.quinncurtis.chart2djava.ChartText
RTTextFrame



*The raised light blue panels are created using **RT3DFrame** objects.*

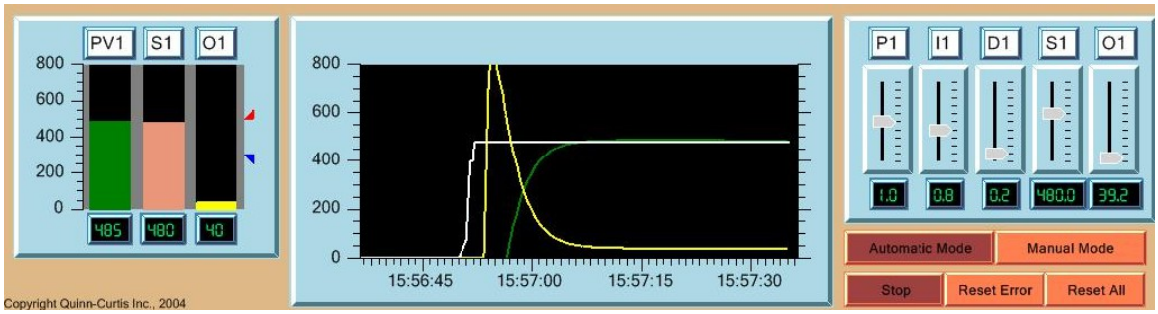
RT3DFrame

This class is used to draw 3D borders and provide the background for many of the other graph objects, most noticeably the **RTPanelMeter** classes. It can also be used directly in your program to provide 3D frames the visually group objects together in a faceplate format.



*The border rectangles in the top graph and the border circle in the bottom graph were created using **RTGenShape** objects.*

RTGenShape This class is used to draw filled and unfilled rectangles, rectangles with rounded corners, general ellipses and aspect ratio corrected circles. These shapes can be used by the programmer to add visual enhancements to graphs.



A complete PID Control tuning center can be created using the PID control tools, bar indicators, scroll frames, buttons and track bars.

RTPIDControl This class represents a simple control loop with support for proportional, integral and derivative control. It includes advanced features for anti-reset windup, error term smoothing, error term reset and rate limiting of control outputs.

RTSymbol This class is used by the **RTAlarmIndicator** class to draw the alarm indicator symbols.

RTTextFrame	This adds a 3D border to the standard QCChart2D **ChartText** text object and recalculates justification parameters to take into account the thickness of the border. It is used by the **RTPanelMeter** classes to display text.

3. QCChart2D for Java Class Summary

Chart view class	The chart view class is a **JPanel** subclass that manages the graph objects placed in the graph
Data classes	There are data classes for simple xy and group data types. There are also data classes that handle **GregorianCalendar** date/time data and contour data.
Scale transform classes	The scale transform classes handle the conversion of physical coordinate values to working coordinate values for a single dimension.
Coordinate transform classes	The coordinate transform classes handle the conversion of physical coordinate values to working coordinate values for a parametric (2D) coordinate system.
Attribute class	The attribute class encapsulates the most common attributes (line color, fill color, line style, line thickness, etc.) for a chart object.
Auto-Scale classes	The coordinate transform classes use the auto-scale classes to establish the minimum and maximum values used to scale a 2D coordinate system. The axis classes also use the auto-scale classes to establish proper tick mark spacing values.
Charting object classes	The chart object classes includes all objects placeable in a chart. That includes axes, axes labels, plot objects (line plots, bar graphs, scatter plots, etc.), grids, titles, backgrounds, images and arbitrary shapes.
Mouse interaction classes	These classes, directly and indirectly derived from the Java **MouseInputListener** class, trap mouse events and permit the user to create and move data cursors, move plot objects, display tooltips and select data points in all types of graphs.
File and printer rendering	These classes render the chart image to a printer, to a JPEG file, or to a Java **Image** object.
Miscellaneous utility classes	Other classes use these for data storage, file I/O, and data processing.

A summary of each category appears in the following section.

Chart Window Classes

javax.swing.JPanel
    ChartView

The starting point of a chart is the **ChartView** class. The **ChartView** class derives from the Java Swing **JPanel** class**,** where the **JPanel** class is the base class for the Swing collection of standard components such as menus, buttons, check boxes, etc. The **ChartView** class manages a collection of chart objects in a chart and automatically updates the chart objects when the underlying window processes a paint event. Since the **ChartView** class is a subclass of the **JPanel** class, it acts as a container for other Java components too. Use a layout manager (including the null layout manager) to position Java components in the **ChartView** window.

Data Classes

ChartDataset
    SimpleDataset
        TimeSimpleDataset
        ElapsedTimeSimpleDataset
        ContourDataset
    GroupDataset
        TimeGroupDataset
        ElapsedTimeGroupDataset

The dataset classes organize the numeric data associated with a plotting object. There are two major types of data supported by the **ChartDataset** class. The first is simple xy data, where for every x-value there is one y-value. The second data type is group data, where every x-value can have one or more y-values. Variants of the simple and group dataset types support time/date **GregorianCalendar** values for the x-values of the dataset.

ChartDataset	The abstract base class for the other dataset classes. It contains data common to all of the dataset classes, such as the x-value array, the number of x-values, the dataset name and the dataset type.
SimpleDataset	Represents simple xy data, where for every x-value there is one y-value.

TimeSimpleDataset A subclass of **SimpleDataset**, it uses **GregorianCalendar** dates as the x-values, and floating-point numbers as the y-values.

ElapsedTimeSimpleDataset A subclass of **SimpleDataset**, it is initialized with **ChartTimeSpan** objects, or milliseconds, in place of the x- or y-values.

ContourDataset A subclass of **SimpleDataset**, it adds a third dimension (z-values) to the x- and y- values of the simple dataset.

GroupDataset Represents group data, where every x-value can have one or more y-values.

TimeGroupDataset A subclass of **GroupDataset**, it uses **GregorianCalendar** dates as the x-values, and floating-point numbers as the y-values.

ElapsedTimeGroupDataset A subclass of **GroupDataset**, it uses **ChartTimeSpan** objects, or milliseconds, as the x-values, and floating point numbers as the y-values.

Scale Classes

ChartScale
- **LinearScale**
- **LogScale**
- **TimeScale**
- **ElapsedTimeScale**

The **ChartScale** abstract base class defines coordinate transformation functions for a single dimension. It is useful to be able to mix and match different scale transform functions for x- and y-dimensions of the **PhysicalCoordinates** class. The job of a **ChartScale** derived object is to convert a dimension from the current *physical* coordinate system into the current *working* coordinate system.

LinearScale A concrete implementation of the **ChartScale** class. It converts a linear physical coordinate system into the working coordinate system.

LogScale A concrete implementation of the **ChartScale** class. It converts a logarithmic physical coordinate system into the working coordinate system.

TimeScale A concrete implementation of the **ChartScale** class. It converts a date/time physical coordinate system into the working coordinate system.

ElapsedTimeScale A concrete implementation of the **ChartScale** class. converts an elapsed time coordinate system into the working coordinate system.

Coordinate Transform Classes

UserCoordinates
WorldCoordinates
WorkingCoordinates
PhysicalCoordinates
CartesianCoordinates
PolarCoordinates
TimeCoordinates

The coordinate transform classes maintain a 2D coordinate system. Many different coordinate systems are used to position and draw objects in a graph. Examples of some of the coordinate systems include the device coordinates of the current window, normalized coordinates for the current window and plotting area, and scaled physical coordinates of the plotting area.

UserCoordinates This class manages the interface to the **Graphics2D** classes and contains routines for drawing lines, rectangles and text using Java device coordinates.

WorldCoordinates This class derives from the **UserCoordinates** class and maps a device independent world coordinate system on top of the Java device coordinate system.

WorkingCoordinates

This class derives from the **WorldCoordinates** class and extends the physical coordinate system of the *plot area* (the area typically bounded by the charts axes) to include the

complete *graph area* (the area of the chart outside of the *plot area*).

PhysicalCoordinates This class is an abstract base class derived from **WorkingCoordinates** and defines the routines needed to map the physical coordinate system of a plot area into a working coordinate system. Different scale objects (**ChartScale** derived) are installed for converting physical x- and y-coordinate values into working coordinate values.

CartesianCoordinates

This class is a concrete implementation of the **PhysicalCoordinates** class and implements a coordinate system used to plot linear, logarithmic and semi-logarithmic graphs.

TimeCoordinates This class is a concrete implementation of the **PhysicalCoordinates** class and implements a coordinate system used to plot **GregorianCalendar** time-based data.

ElapsedTimeCoordinates This class is a subclass of the **CartesianCoordinates** class and implements a coordinate system used to plot elapsed time data.

PolarCoordinates This class is a concrete implementation of the **PhysicalCoordinates** class and implements a coordinate system used to plot polar coordinate data.

AntennaCoordinates This class is a subclass of the **CartesianCoordinates** class and implements a coordinate system used to plot antenna coordinate data. The antenna coordinate system differs from the more common polar coordinate system in that the radius can have plus/minus values, the angular values are in degrees, and the angular values increase in the clockwise direction.

Attribute Class

ChartAttribute
ChartGradient

This class consolidates the common line and fill attributes as a single class. Most of the graph objects have a property of this class that controls the color, line thickness and fill attributes of the object. The **ChartGradient** class expands the number of color options available in the **ChartAttribute** class.

ChartAttribute	This class consolidates the common line and fill attributes associated with a **GraphObj** object into a single class.
ChartGradient	A **ChartGradient** can be added to a **ChartAttribute** object, defining a multicolor gradient that is applied wherever the color fill attribute is normally used

Auto-Scaling Classes

AutoScale
- **LinearAutoScale**
- **LogAutoScale**
- **TimeAutoScale**
- **ElapsedTimeAutoScale**

Usually, programmers do not know in advance the scale for a chart. Normally the program needs to analyze the current data for minimum and maximum values and create a chart scale based on those values. Auto-scaling, and the creation of appropriate axes, with endpoints at even values, and well-rounded major and minor tick mark spacing, is quite complicated. The **AutoScale** classes provide tools that make automatic generation of charts easier.

AutoScale	This class is the abstract base class for the auto-scale classes.
LinearAutoScale	This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the numeric values in **SimpleDataset** and **GroupDataset** objects. Linear scales and axes use it for auto-scale calculations.
LogAutoScale	This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the numeric values in **SimpleDataset** and **GroupDataset** objects. Logarithmic scales and axes use it for auto-scale calculations.

TimeAutoScale This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the **GregorianCalendar** values in **TimeSimpleDataset** and **TimeGroupDataset** objects. Date/time scales and axes use it for auto-scale calculations.

ElapsedTimeAutoScale This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the numeric values in **ElapsedTimeSimpleDataset** and **ElapsedTimeGroupDataset** objects. The elapsed time classes use it for auto-scale calculations.

Chart Object Classes

Chart objects are graph objects that can be rendered in the current graph window. This is in comparison to other classes that are purely calculation classes, such as the coordinate conversion classes. All chart objects have certain information in common. This includes instances of **ChartAttribute** and **PhysicalCoordinates** classes. The **ChartAttribute** class contains basic color and line style information for the object, while the **PhysicalCoordinates** maintains the coordinate system used by object. The majority of classes in the library derive from the **GraphObj** class, each class a specific charting object such as an axis, an axis label, a simple plot or a group plot. Add **GraphObj** derived objects (axes, plots, labels, title, etc.) to a graph using the **ChartView.addChartObject** method.

GraphObj This class is the abstract base class for all drawable graph objects. It contains information common to all chart objects. This class includes references to instances of the **ChartAttribute** and **PhysicalCoordinates** classes. The **ChartAttribute** class contains basic color and line style information for the object, while the **PhysicalCoordinates** maintains the coordinate system used by object. The majority of classes in the library derive from the **GraphObj** class, each class a specific charting object such as an axis, an axis label, a simple plot or a group plot

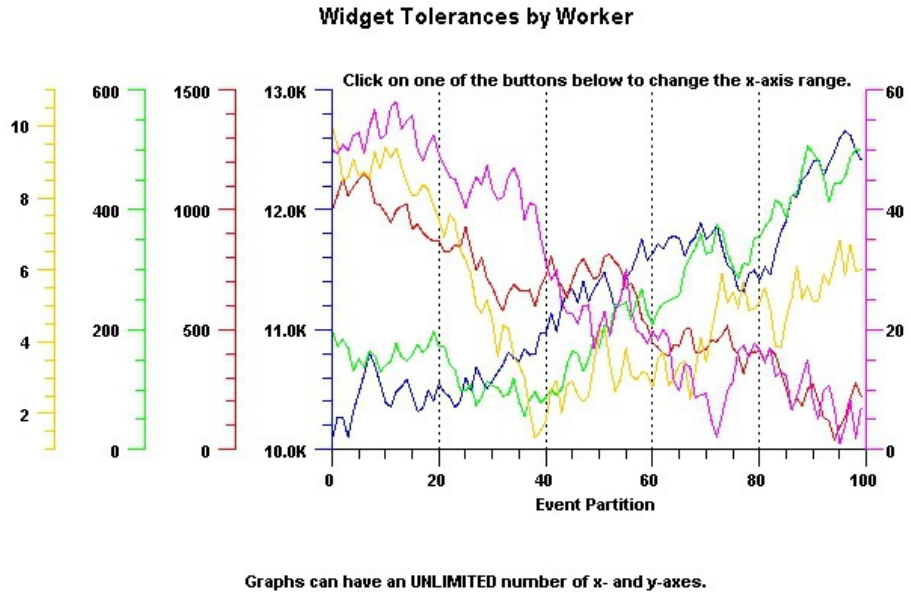
Background This class fills the background of the entire chart, or the plot area of the chart, using a solid color, a color gradient, or a texture.

Axis Classes

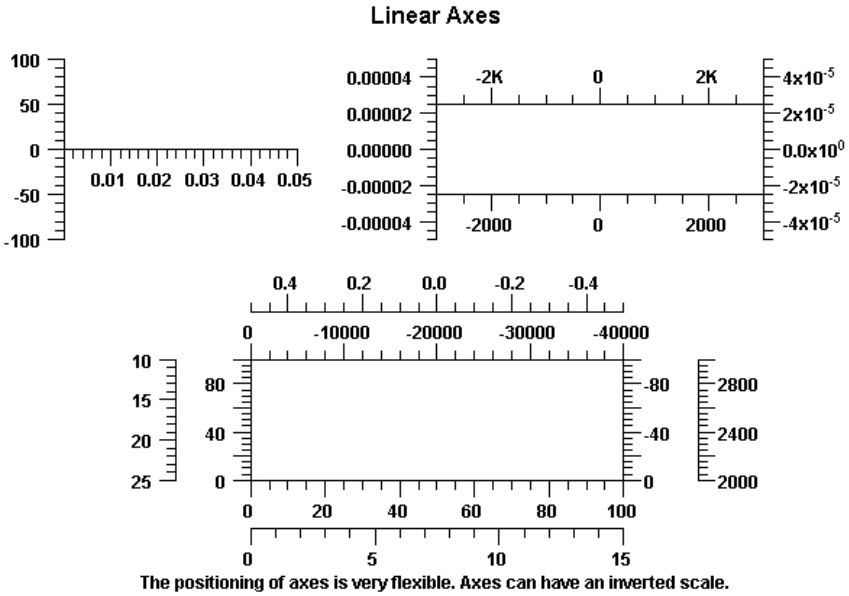
Axis

- **LinearAxis**
  - **PolarAxes**
  - **AntennaAxes**
  - **ElapsedTimeAxis**
- **LogAxis**
- **TimeAxis**

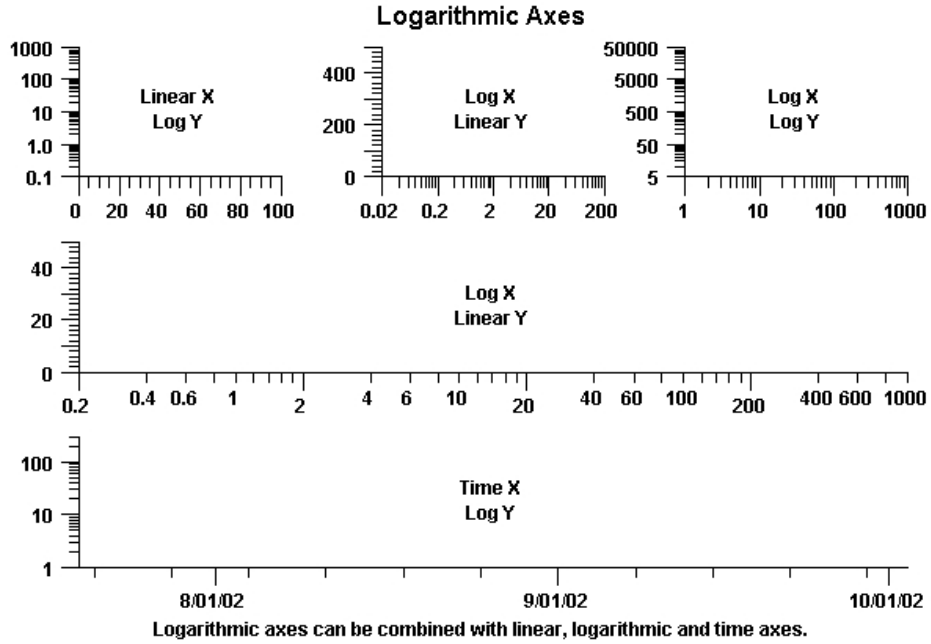
Creating a **PhysicalCoordinates** coordinate system does not automatically create a pair of x- and y-axes. Axes are separate charting objects drawn with respect to a specific **PhysicalCoordinates** object. The coordinate system and the axes do not need to have the same limits. In general, the limits of the coordinate system should be greater than or equal to the limits of the axes. The coordinate system may have limits of 0 to 15, while you may want the axes to extend from 0 to 10.



Axis This class is the abstract base class for the other axis classes. It contains data and drawing routines common to all axis classes.

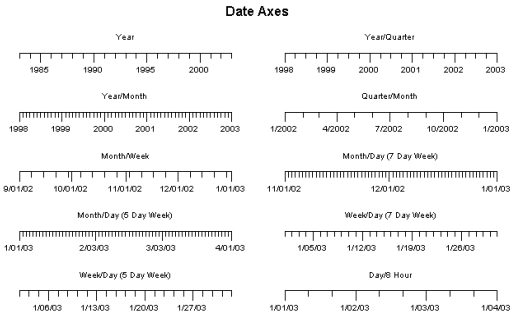


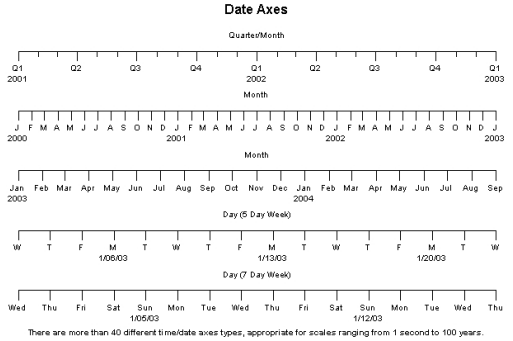
LinearAxis This class implements a linear axis with major and minor tick marks placed at equally spaced intervals.

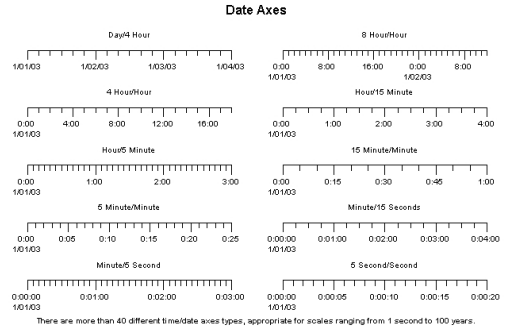


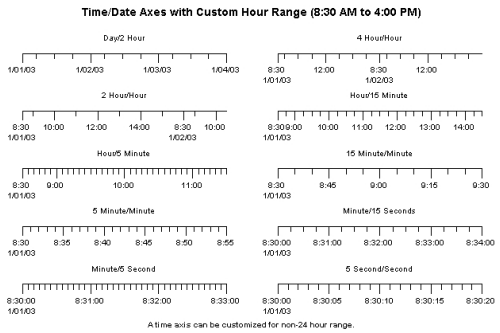
LogAxis

This class implements a logarithmic axis with major tick marks placed on logarithmic intervals, for example 1, 10,100 or 30, 300, 3000. The minor tick marks are placed within the major tick marks using linear intervals, for example 2, 3, 4, 5, 6, 7, 8, 9, 20, 30, 40, 50,.., 90. An important feature of the **LogAxis** class is that the major and minor tick marks do not have to fall on decade boundaries. A logarithmic axis must have a positive range exclusive of 0.0, and the tick marks can represent any logarithmic scale.



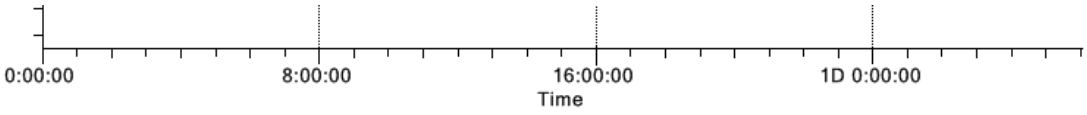






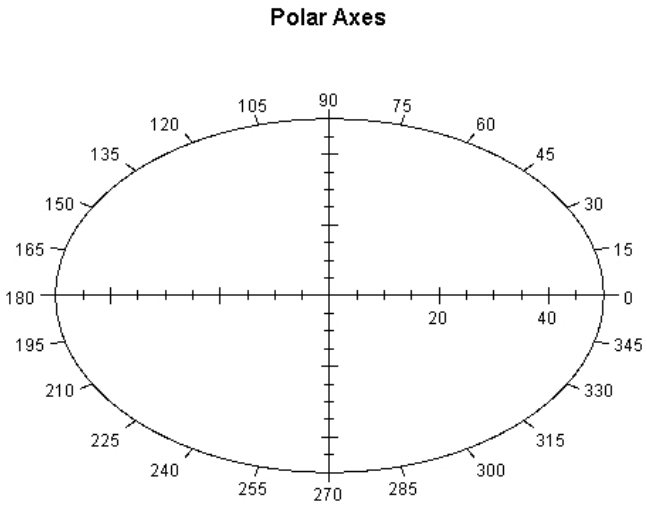
TimeAxis

This class is the most complex of the axis classes. It supports time scales ranging from 1 millisecond to hundreds of years. Dates and times are specified using the **GregorianCalendar** class. The major and minor tick marks can fall on any time base, where a time base represents seconds, minutes, hours, days, weeks, months or years. The scale can exclude weekends, for example, Friday, October 20, 2000 is immediately followed by Monday, October 23, 2000. A day can also have a custom range, for example a range of 9:30 AM to 4:00 PM. The chart time axis excludes time outside of this range. This makes the class very useful for the inter-day display of financial market information (stock, bonds, commodities, options, etc.) across several days, months or years.



ElapsedTimeAxis

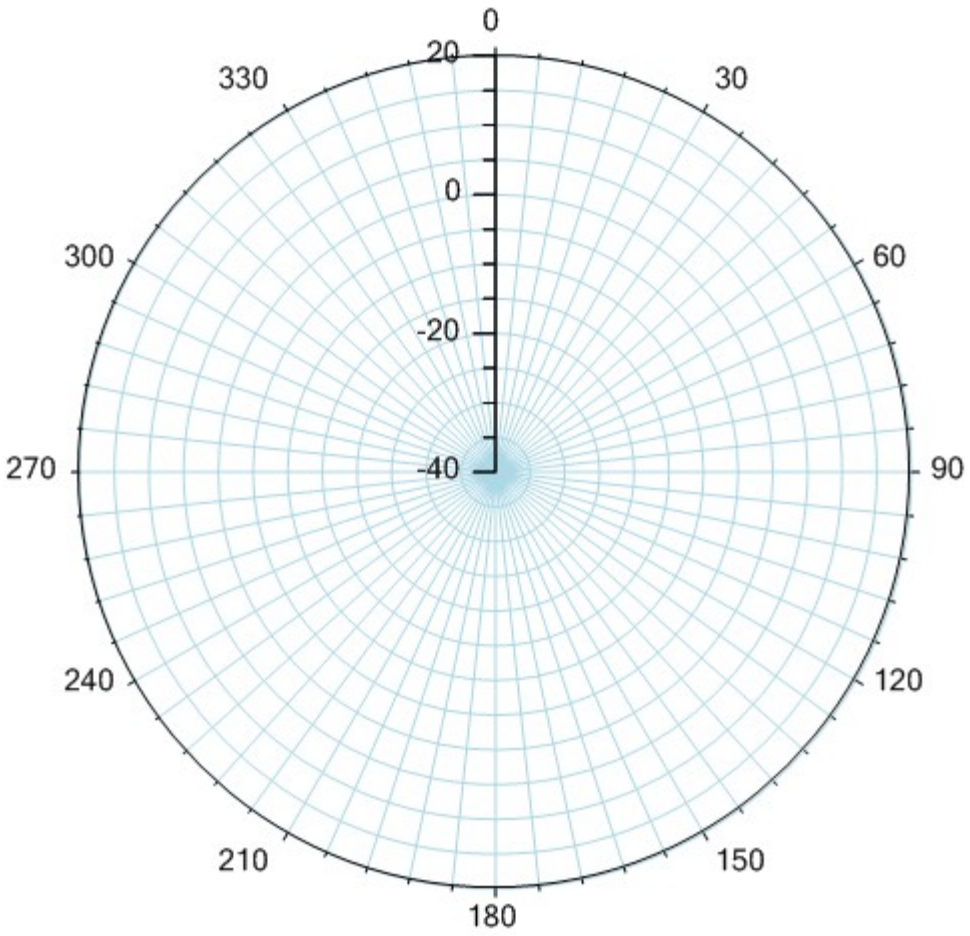
The elapsed time axis is very similar to the linear axis and is subclassed from that class. The main difference is the major and minor tick mark spacing calculated by the CalcAutoAxis method takes into account the base 60 of seconds per minute and minutes per hour, and the base 24 of hours per day. It is a continuous linear scale.



A polar axis consists of the x and y axis for magnitude, and the outer circle for the angle.

PolarAxes

This class has three separate axes: two linear and one circular. The two linear axes, scaled for +- the magnitude of the polar scale, form a cross with the center of both axes at the origin (0, 0. The third axis is a circle centered on the origin with a radius equal to the magnitude of the polar scale. This circular axis represents 360 degrees (or 2 Pi radians) of the polar scale and the tick marks that circle this axis are spaced at equal degree intervals.



AntennaAxes This class has two axes: one linear y-axis and one circular axis. The linear axis is scaled for the desired range of radius values. This can extend from minus values to plus values. The second axis is a circle centered on the origin with a radius equal to the range of the radius scale. This circular axis represents 360 degrees of the antenna scale and the tick marks that circle this axis are spaced at equal degree intervals.

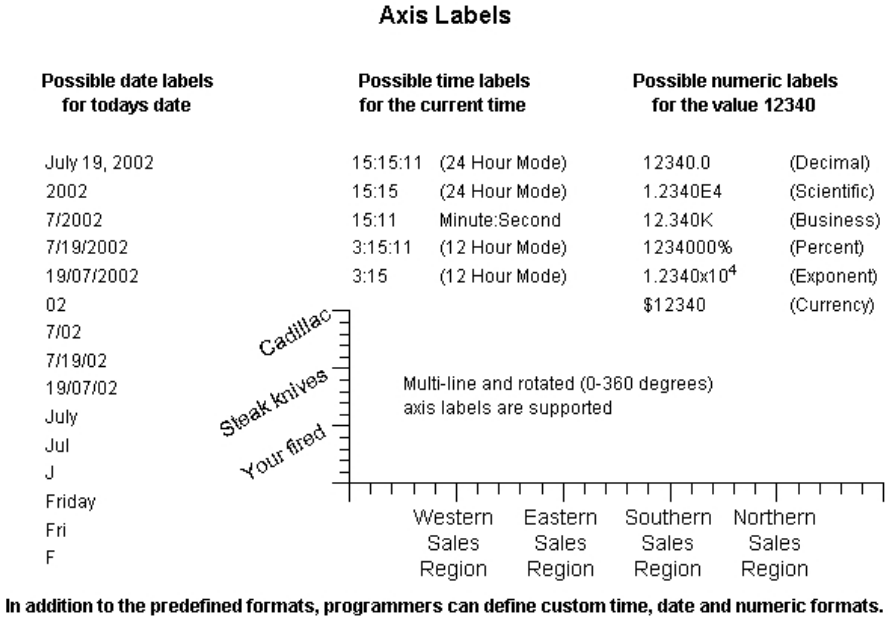
Axis Label Classes

AxisLabels

- **NumericAxisLabels**
- **StringAxisLabels**
- **PolarAxesLabels**
- **AntennaAxesLabels**

TimeAxisLabels
ElapsedTimeAxisLabels

Axis labels inform the user of the x- and y-scales used in the chart. The labels center on the major tick marks of the associated axis. Axis labels are usually numbers, times, dates, or arbitrary strings.



AxisLabels — This class is the abstract base class for all axis label objects. It places numeric labels, date/time labels, or arbitrary text labels, at the major tick marks of the associated axis object. In addition to the standard font options (type, size, style, color, etc.), axis label text can be rotated 360 degrees in one degree increments.

NumericAxisLabels — This class labels the major tick marks of the **LinearAxis**, and **LogAxis** classes. The class supports many predefined and user-definable formats, including numeric, exponent, percentage, business and currency formats.

StringAxisLabels — This class labels the major tick marks of the **LinearAxis**, and **LogAxis** classes using user-defined strings.

TimeAxisLabels This class labels the major tick marks of the associated **TimeAxis** object. The class supports many time (23:59:59) and date (5/17/2001) formats. It is also possible to define custom date/time formats.

ElapsedTimeAxisLabels This class labels the major tick marks of the associated **ElapsedTimeAxis** object. The class supports HH:MM:SS and MM:SS formats, with decimal seconds out to 0.00001, i.e. "12:22:43.01234". It also supports a cumulative hour format (101:51:22), and a couple of day formats (4.5:51:22, 4D 5:51:22).

PolarAxesLabels This class labels the major tick marks of the associated **PolarAxes** object. The x-axis is labeled from 0.0 to the polar scale magnitude, and the circular axis is labeled counter clockwise from 0 to 360 degrees, starting at 3:00.

AntennaAxesLabels This class labels the major tick marks of the associated **AntennaAxes** object. The y-axis is labeled from the radius minimum to the radius maximum. The circular axis is labeled clockwise from 0 to 360 degrees, starting at 12:00.

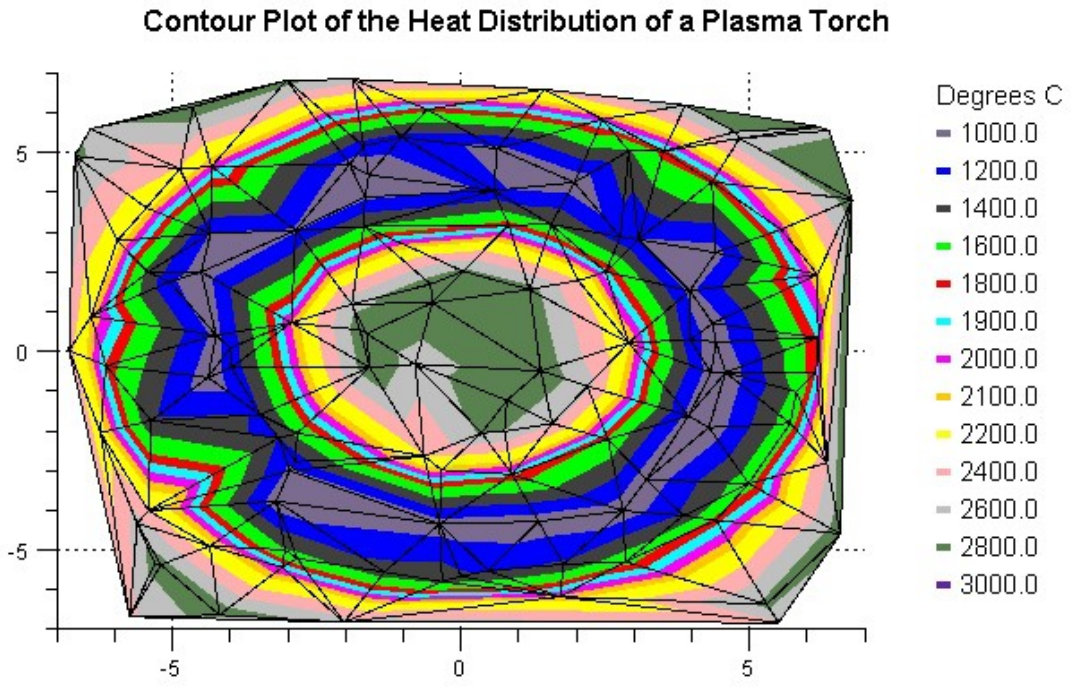
Chart Plot Classes

ChartPlot
- **ContourPlot**
- **GroupPlot**
- **PieChart**
- **PolarPlot**
- **AntennaPlot**
- **SimplePlot**

Plot objects are objects that display data organized in a **ChartDataset** class. There are five main categories: simple, group, polar, contour and pie plots. Simple plots graph data organized as a simple set of xy data points. The most common examples of simple plots are line plots, bar graphs and scatter plots. Group plots graph data organized as multiple y-values for each x-value. The most common examples of group plots are stacked bar graphs, open-high-low-close plots, and candlestick plots. Polar charts plot data organized as a simple set of data points, where each data point represents a polar magnitude and angle pair, rather than xy Cartesian coordinate values. The most common example of polar charts is the display of complex numbers (a + b*i*), and it is used in many

engineering disciplines. The contour plot type displays the iso-lines, or contours, of a 3D surface using either lines or regions of solid color. The last plot object category is the pie chart, were a pie wedge represents each data value. The size of the pie wedge is proportional to the fraction (data value / sum of all data values).

ChartPlot This class is the abstract base class for chart plot objects. It contains a reference to a **ChartDataset** derived class containing the data associated with the plot.



ContourPlot This class is a concrete implementation of the **ChartPlot** class and displays a contour plot using either lines, or regions filled with color.

Group Plot Classes

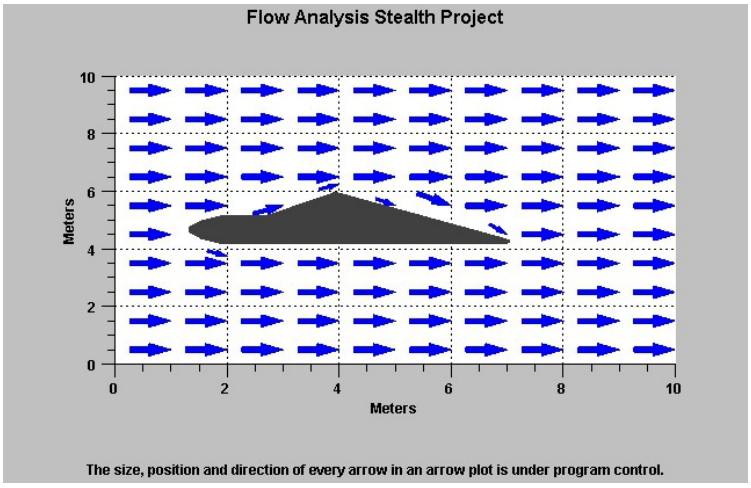
GroupPlot

- **ArrowPlot**
- **BoxWhiskerPlot**

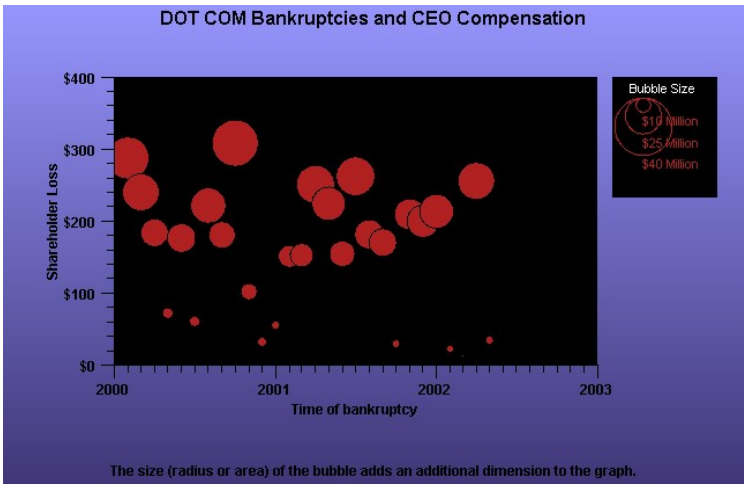
BubblePlot
CandlestickPlot
CellPlot
ErrorBarPlot
FloatingBarPlot
FloatingStackedBarPlot
GroupBarPlot
GroupVersaPlot
HistogramPlot
LineGapPlot
MultiLinePlot
OHLCPlot
StackedBarPlot
StackedLinePlot
GroupVeraPlot

Group plots use data organized as arrays of x- and y-values, where there is one or more y for every x.. Group plot types include multi-line plots, stacked line plots, stacked bar plots, group bar plots, error bar plots, floating bar plots, open-high-low-close plots, candlestick plots, arrow plots, histogram plots, cell plots and bubble plots.

GroupPlot This class is an abstract base class for all group plot classes.

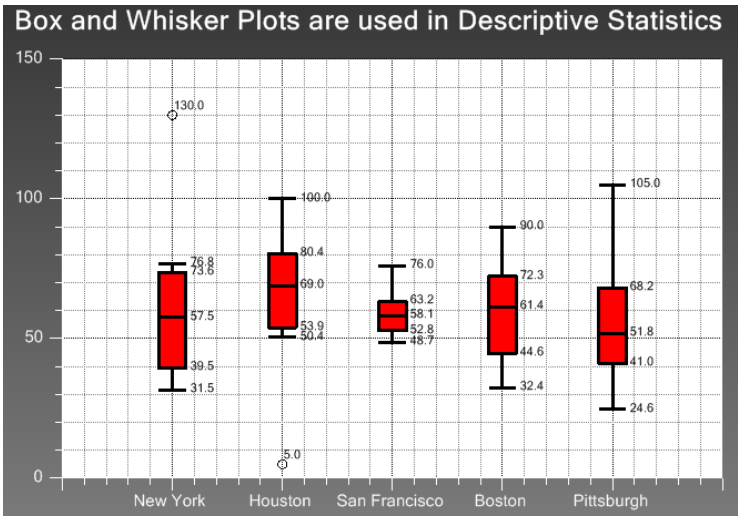


ArrowPlot This class is a concrete implementation of the **GroupPlot** class and it displays a collection of arrows as defined by the data in a group dataset. The position, size, and rotation of each arrow in the collection is independently controlled



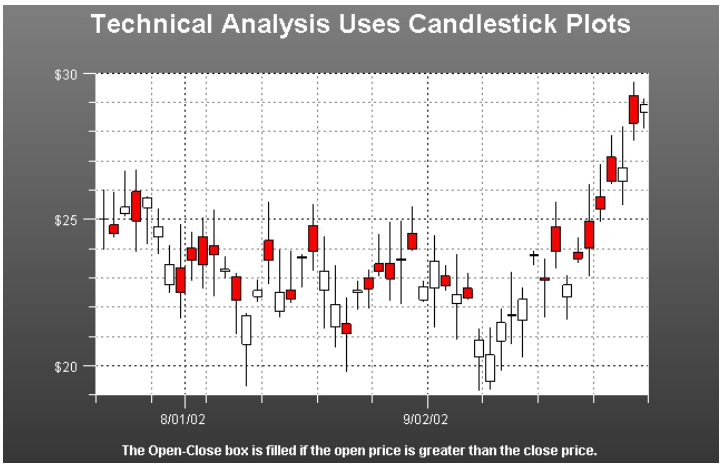
BubblePlot

This class is a concrete implementation of the **GroupPlot** class and displays bubble plots. The values in the dataset specify the position and size of each bubble in a bubble chart.

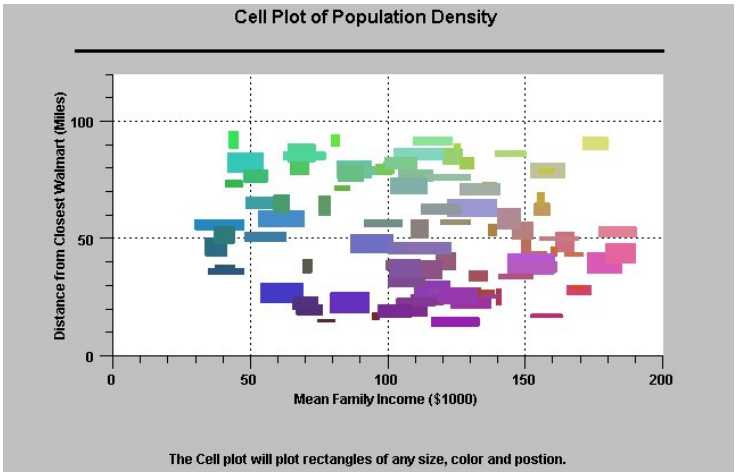


BoxWhiskerPlot

This class is a concrete implementation of the **GroupPlot** class and displays box and whisker plots. The BoxWhiskerPlot class graphically depicts groups of numerical data through their five-number summaries (the smallest observation, lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation).

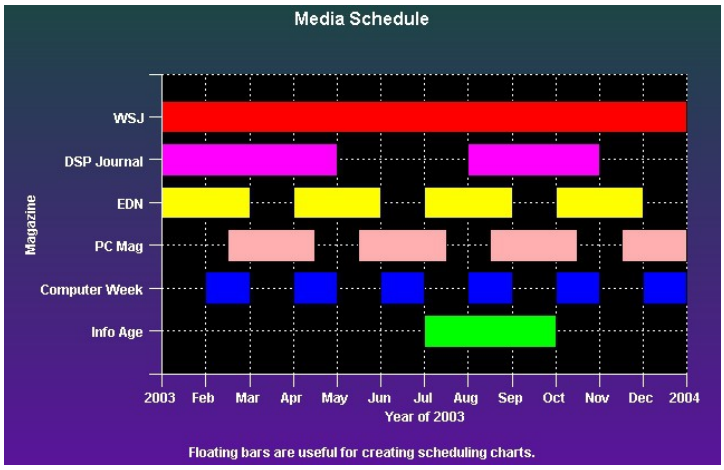


CandlestickPlot This class is a concrete implementation of the **GroupPlot** class and displays stock market data in an open-high-low-close format common in financial technical analysis.

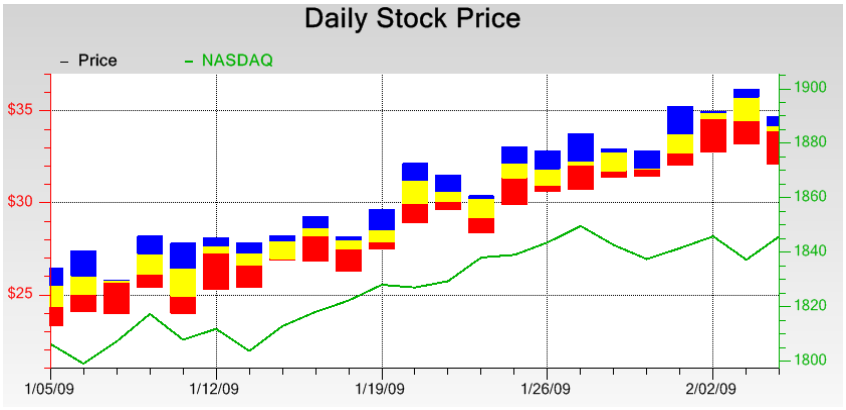


CellPlot This class is a concrete implementation of the **GroupPlot** class and displays cell plots. A cell plot is a collection of rectangular objects with independent positions, widths and heights, specified using the values of the associated group dataset.

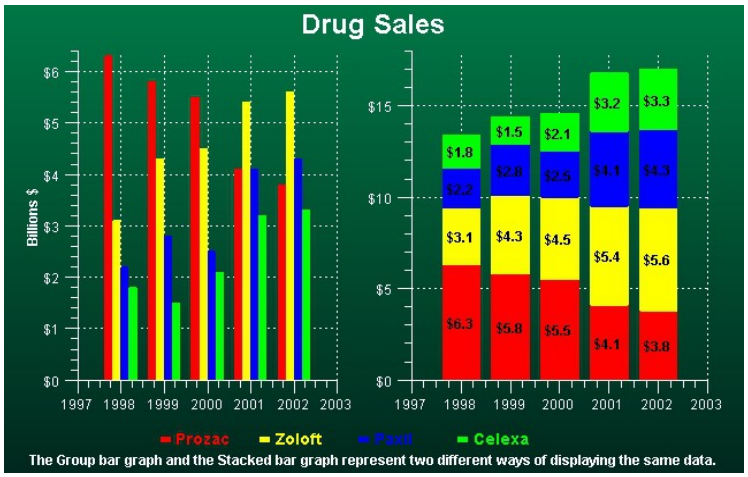
ErrorBarPlot This class is a concrete implementation of the **GroupPlot** class and displays error bars. Error bars are two lines positioned about a data point that signify the statistical error associated with the data point



FloatingBarPlot This class is a concrete implementation of the **GroupPlot** class and displays free-floating bars in a graph. The bars are free floating because each bar does not reference a fixed base value, as do simple bar plots, stacked bar plots and group bar plots.

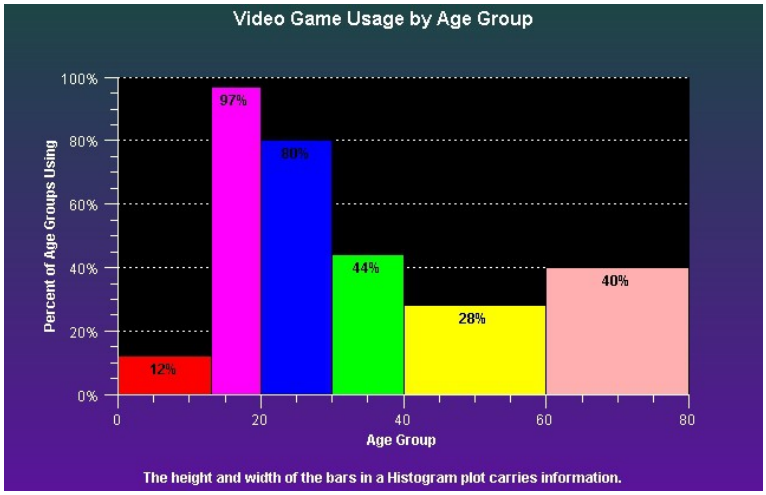


FloatingStackedBarPlot This class is a concrete implementation of the **GroupPlot** class and displays free-floating stacked bars. The bars are free floating because each bar does not reference a fixed base value, as do simple bar plots, stacked bar plots and group bar plots.



GroupBarPlot

This class is a concrete implementation of the **GroupPlot** class and displays group data in a group bar format. Individual bars, the height of which corresponds to the group y-values of the dataset, display side by side, as a group, justified with respect to the x-position value for each group. The group bars share a common base value.



GroupVeraPlot

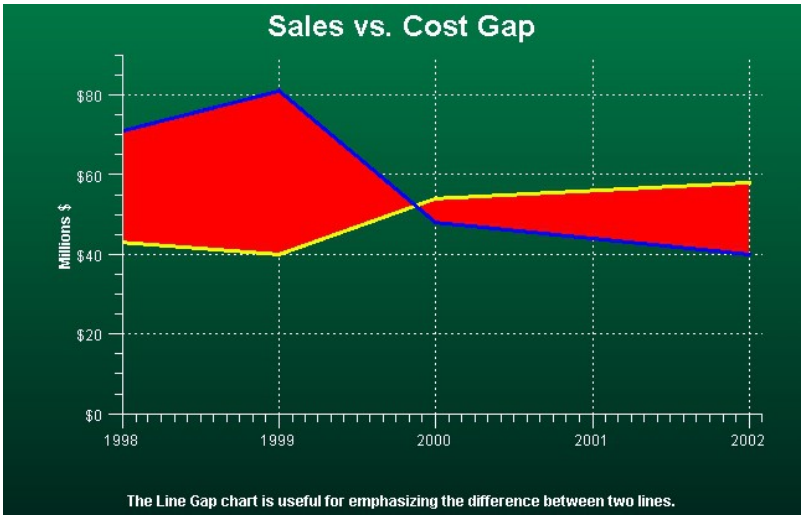
The **GroupVersaPlot** is a plot type that can be any of the eight group plot types: GROUPBAR, STACKEDBAR, CANDLESTICK, OHLC, MULTILINE, STACKEDLINE, FLOATINGBAR and FLOATING_STACKED_BAR. Use it when you want to be able to change from one plot type to another, without deleting the instance of the old plot object and creating an instance of the new.

StackedBarPlot

This class is a concrete implementation of the **GroupPlot** class and displays data as stacked bars. In a stacked bar plot each group is stacked on top of one another, each group bar a cumulative sum of the related group items before it.

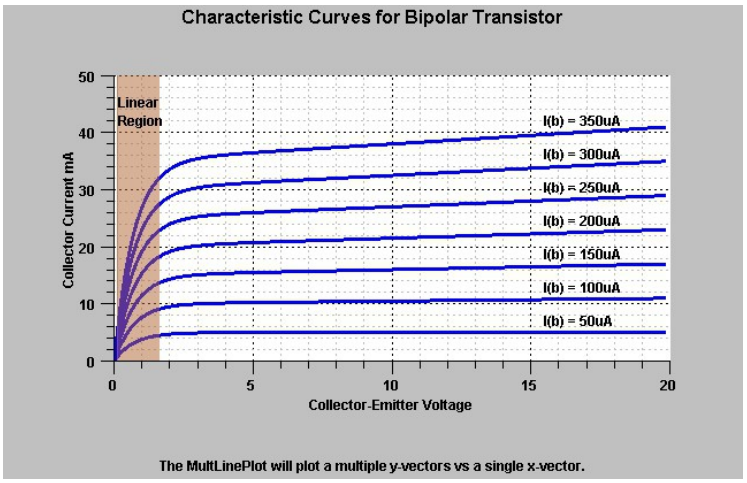
HistogramPlot

This class is a concrete implementation of the **GroupPlot** class and displays histogram plots. A histogram plot is a collection of rectangular objects with independent widths and heights, specified using the values of the associated group dataset. The histogram bars share a common base value.



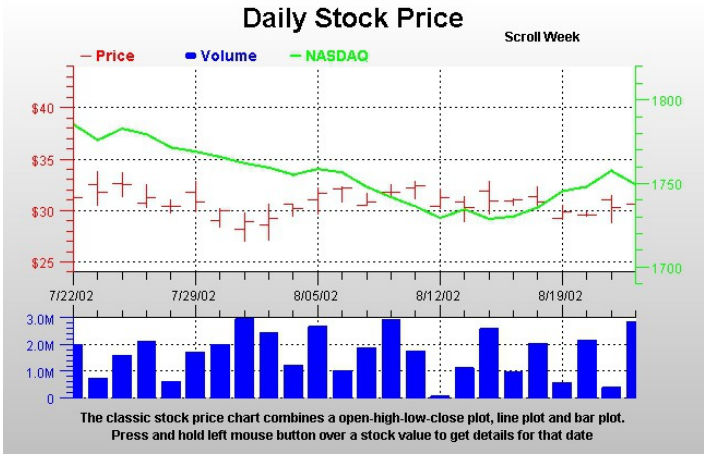
LineGapPlot

This class is a concrete implementation of the **GroupPlot** class. A line gap chart consists of two lines plots where a contrasting color fills the area between the two lines, highlighting the difference.



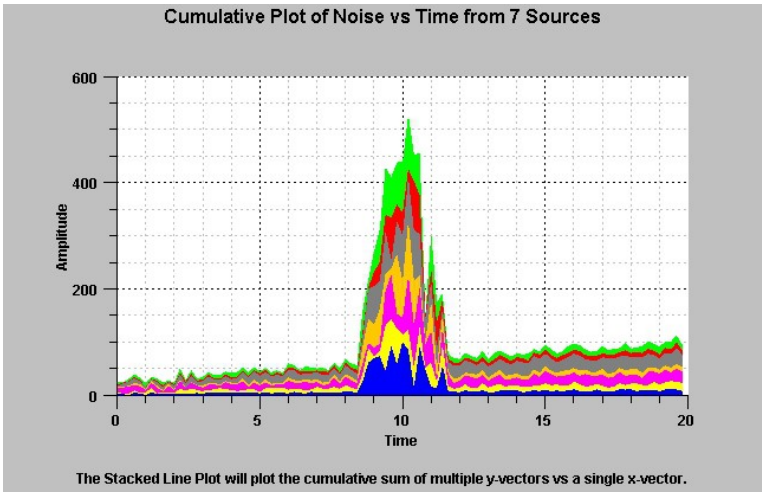
MultiLinePlot

This class is a concrete implementation of the **GroupPlot** class and displays group data in multi-line format. A group dataset with four groups will display four separate line plots. The y-values for each line of the line plot represent the y-values for each group of the group dataset. Each line plot share the same x-values of the group dataset.



OHLCPlot

This class is a concrete implementation of the **GroupPlot** class and displays stock market data in an open-high-low-close format common in financial technical analysis. Every item of the plot is a vertical line, representing High and Low values, with two small horizontal "flags", one left and one right extending from the vertical High-Low line and representing the Open and Close values.



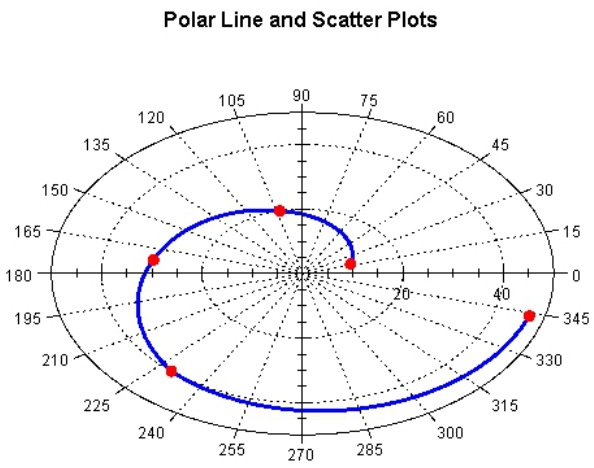
StackedLinePlot This class is a concrete implementation of the **GroupPlot** class and displays data in a stacked line format. In a stacked line plot each group is stacked on top of one another, each group line a cumulative sum of the related group items before it.

Polar Plot Classes

PolarPlot
- **PolarLinePlot**
- **PolarScatterPlot**

Polar plots that use data organized as arrays of x- and y-values, where an x-value represents the magnitude of a point in polar coordinates, and the y-value represents the angle, in radians, of a point in polar coordinates. Polar plot types include line plots and scatter plots.

PolarPlot This class is an abstract base class for the polar plot classes.



PolarLinePlot This class is a concrete implementation of the **PolarPlot** class and displays data in a simple line plot format. The lines drawn between adjacent data points use polar coordinate interpolation.

PolarScatterPlot This class is a concrete implementation of the **PolarPlot** class and displays data in a simple scatter plot format.

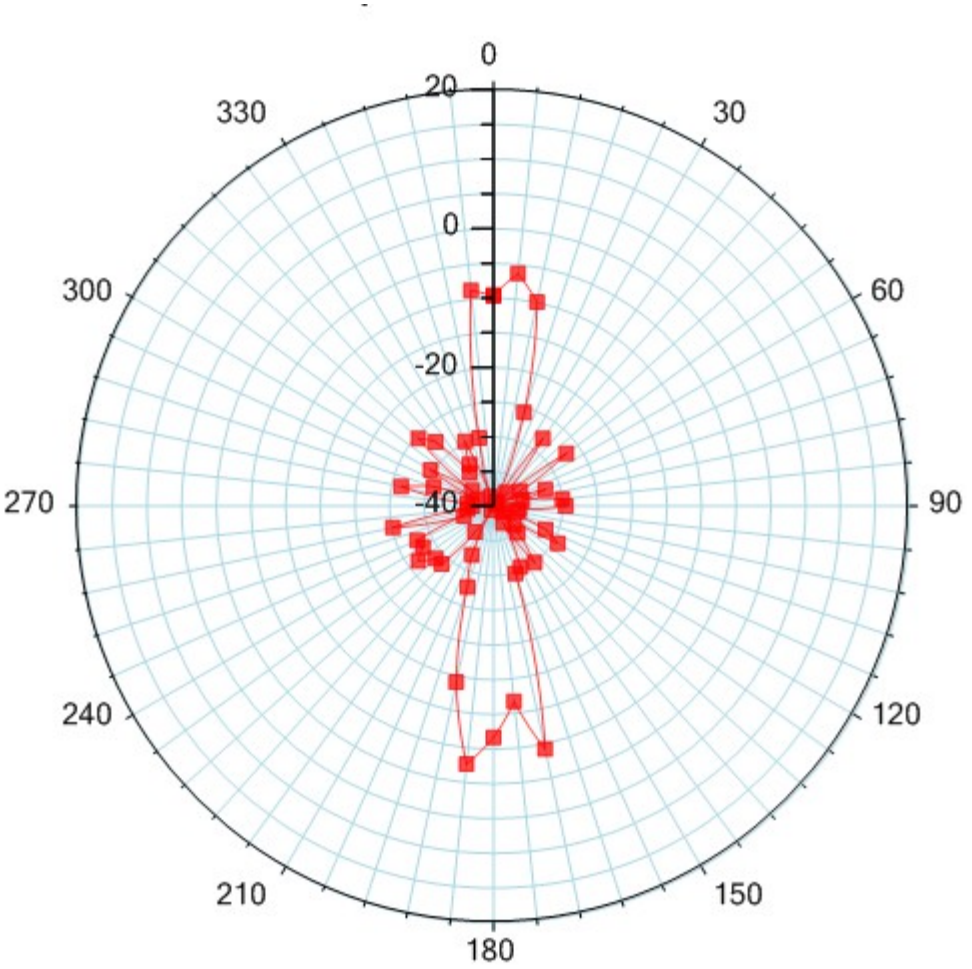
Antenna Plot Classes

AntennaPlot
- **AntennaLinePlot**
- **AntennaScatterPlot**
- **AntennaLineMarkerPlot**

GraphObj
- **AntennaAnnotation**

Antenna plots that use data organized as arrays of x- and y-values, where an x-value represents the radial value of a point in antenna coordinates, and the y-value represents the angle, in degrees, of a point in antenna coordinates. Antenna plot types include line plots, scatter plots, line marker plots, and an annotation class.

AntennaPlot This class is an abstract base class for the polar plot classes.



AntennaLineMarkerPlot

AntennaLinePlot This class is a concrete implementation of the **AntennaPlot** class and displays data in a simple line plot format. The lines drawn between adjacent data points use antenna coordinate interpolation.

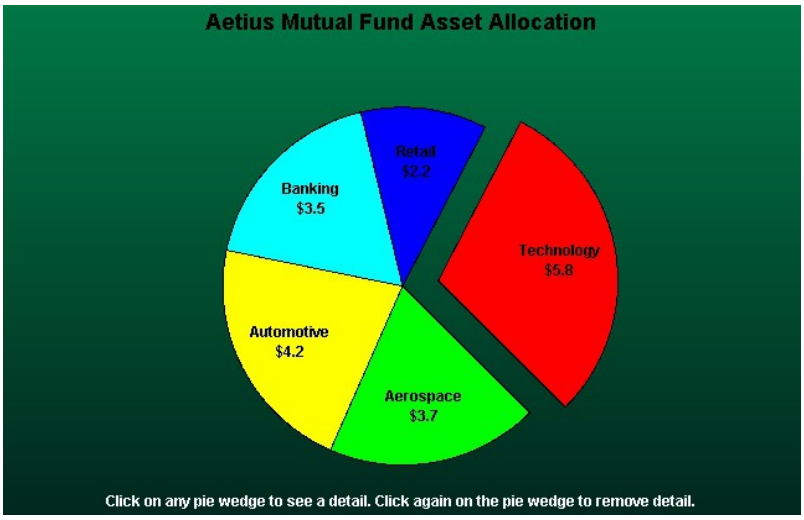
AntennaScatterPlot This class is a concrete implementation of the **AntennaPlot** class and displays data in a simple scatter plot format.

AntennaLineMarkerPlot This class is a concrete implementation of the **AntennaPlot** class and displays data in a simple line markder plot format.

AntennaAnnotation This class is used to highlight, or mark, a specific attribute of the chart. It can mark a constant radial value using a circle, or it can mark a constant angular value using a radial line from the origin to the outer edge of the scale.

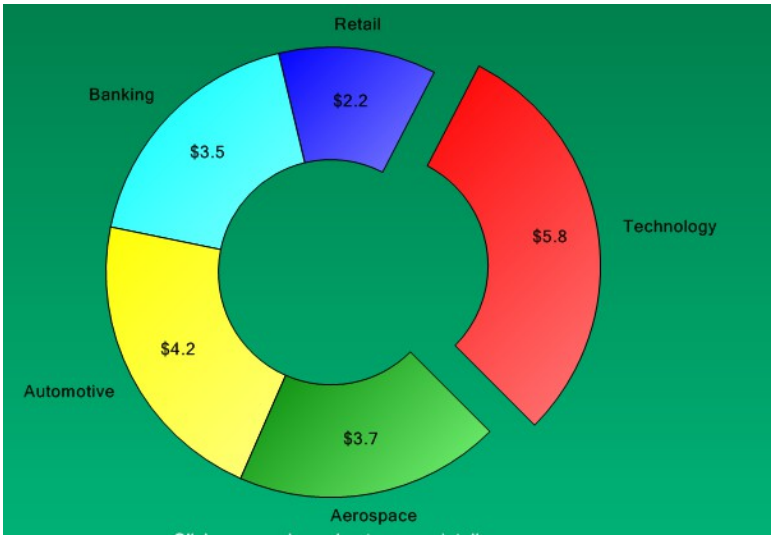
Pie and Ring Chart Classes

It uses data organized as arrays of x- and y-values, where an x-value represents the numeric value of a pie wedge, and a y-value specifies the offset (or "explosion") of a pie wedge with respect to the center of the pie.



PieChart

This class plots data in a simple pie chart format. It uses data organized as arrays of x- and y-values, where an x-value represents the numeric value of a pie wedge, and a y-value specifies the offset (or "explosion") of a pie wedge with respect to the center of the pie.



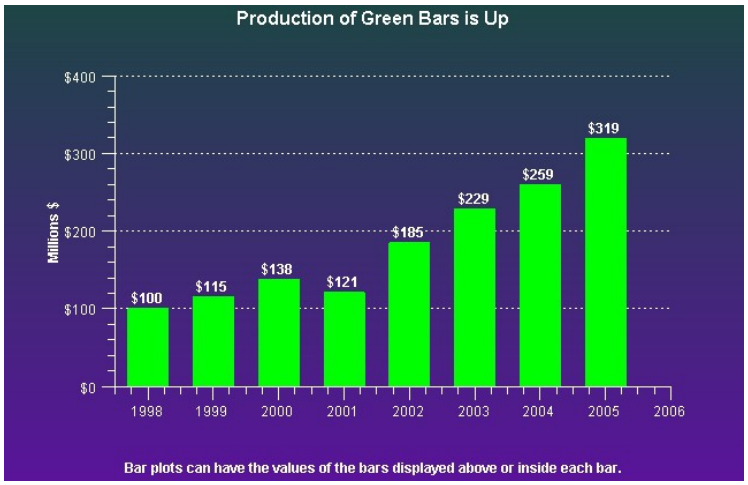
RingChart The ring chart plots data in a modified pie chart format known as a ring chartt. It uses data organized as arrays of x- and y-values, where an x-value represents the numeric value of a ring segment, and a y-value specifies the offset (or "explosion") of a ring segment with respect to the origin of the ring.

Simple Plot Classes

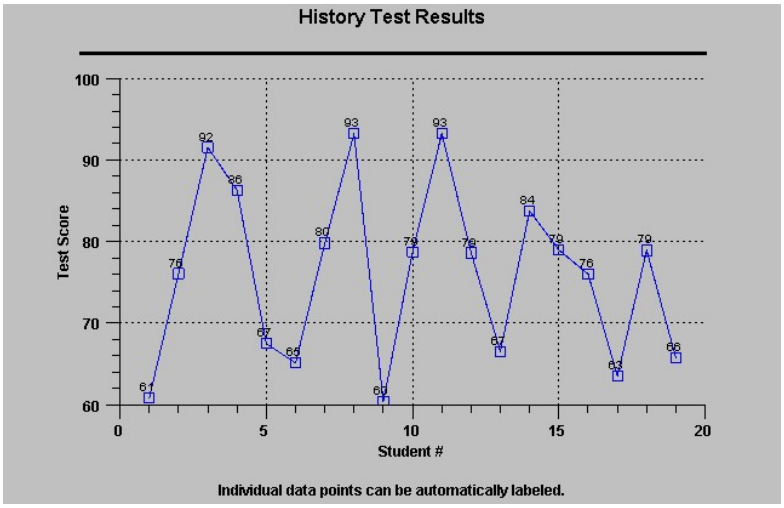
SimplePlot
- **SimpleBarPlot**
- **SimpleLineMarkerPlot**
- **SimpleLinePlot**
- **SimpleScatterPlot**
- **SimpleVeraPlot**

Simple plots use data organized as a simple array of xy points, where there is one y for every x. Simple plot types include line plots, scatter plots, bar graphs, and line-marker plots.

SimplePlot This class is an abstract base class for all simple plot classes.

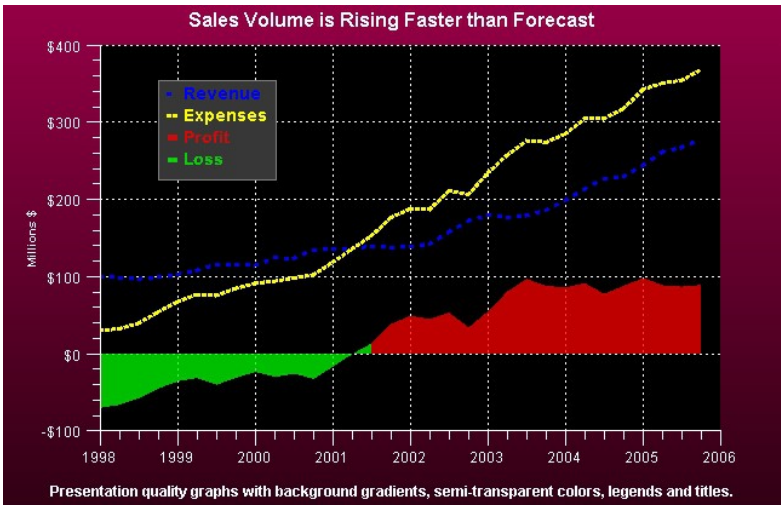


SimpleBarPlot This class is a concrete implementation of the **SimplePlot** class and displays data in a bar format. Individual bars, the maximum value of which corresponds to the y-values of the dataset, are justified with respect to the x-values.



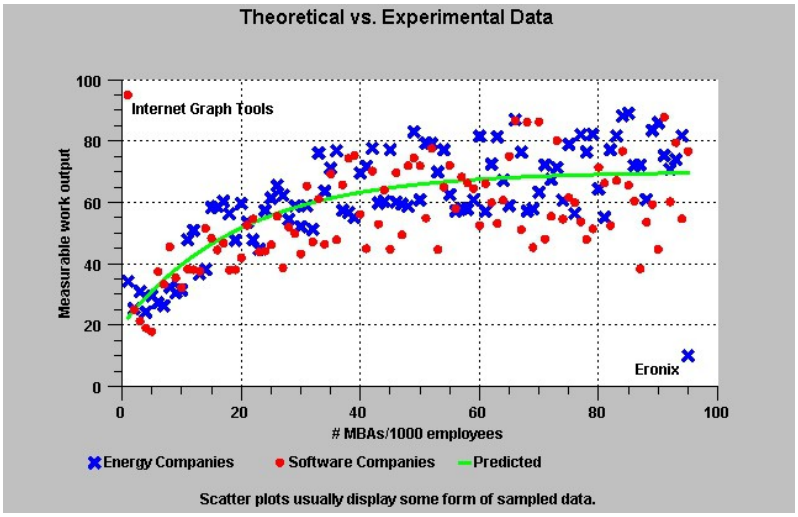
SimpleLineMarkerPlot

This class is a concrete implementation of the **SimplePlot** class and it displays simple datasets in a line plot format where scatter plot symbols highlight individual data points.



SimpleLinePlot

This class is a concrete implementation of the **SimplePlot** class it displays simple datasets in a line plot format. Adjacent data points are connected using a straight, or a step line.



SimpleScatterPlot This class is a concrete implementation of the **SimplePlot** class and it displays simple datasets in a scatter plot format where each data point is represented using a symbol.

SimpleVersaPlot The **SimpleVersaPlot** is a plot type that can be any of the four simple plot types: LINE_MARKER_PLOT, LINE_PLOT, BAR_PLOT, SCATTER_PLOT. It is used when you want to be able to change from one plot type to another, without deleting the instance of the old plot object and creating an instance of the new.

Legend Classes

LegendItem
BubblePlotLegendItem
Legend
    StandardLegend
    BubblePlotLegend

Legends provide a key for interpreting the various plot objects in a graph. It organizes a collection of legend items, one for each plot objects in the graph, and displays them in a rectangular frame.

Legend This class is the abstract base class for chart legends.

LegendItem This class is the legend item class for all plot objects except for bubble plots. Each legend item manages one symbol and descriptive text for that symbol. The **StandardLegend** class uses objects of this type as legend items.

BubblePlotLegendItem

This class is the legend item class for bubble plots. Each legend item manages a circle and descriptive text specifying the value of a bubble of this size. The **BubblePlotLegend** class uses objects of this type as legend items.

StandardLegend This class is a concrete implementation of the **Legend** class and it is the legend class for all plot objects except for bubble plots. The legend item objects display in a row or column format. Each legend item contains a symbol and a descriptive string. The symbol normally associates the legend item to a particular plot object, and the descriptive string describes what the plot object represents.

BubblePlotLegend This class is a concrete implementation of the **Legend** class and it is a legend class used exclusively with bubble plots. The legend item objects display as offset, concentric circles with descriptive text giving the key for the value associated with a bubble of this size.

Grid Classes

Grid
- **PolarGrid**
- **AntennaGrid**

Grid lines are perpendicular to an axis, extending the major and/or minor tick marks of the axis across the width or height of the plot area of the chart.

Grid This class defines the grid lines associated with an axis. Grid lines are perpendicular to an axis, extending the major and/or minor tick marks of the axis across the width or height of the plot area of the chart. This class works in conjunction with the **LinearAxis**, **LogAxis** and **TimeAxis** classes.

PolarGrid This class defines the grid lines associated with a polar axis. A polar chart grid consists of two sets of lines. The first set is a group of concentric circles, centered on the origin and passing through the major and/or minor tick marks of the polar magnitude horizontal and vertical axes. The second set is a group of radial lines, starting at the origin and extending to the outermost edge of the polar plot circle, passing through the major and minor tick marks of the polar angle circular axis. This class works in conjunction with the **PolarAxes** class.

AntennaGrid Analogous to the **PolarGrid**, this class draws radial, and circular grid lines for an Antenna chart.

Chart Text Classes

ChartText
- **ChartTitle**
- **AxisTitle**
- **ChartLabel**
  - **NumericLabel**
  - **TimeLabel**
  - **StringLabel**
  - **ElapsedTimeLabel**

The chart text classes draw one or more strings in the chart window. Different classes support different numeric formats, including floating-point numbers, date/time values and multi-line text strings. International formats for floating-point numbers and date/time values are also supported.

ChartText This class draws a string in the current chart window. It is the base class for the **ChartTitle**, **AxisTitle** and **ChartLabel** classes. The **ChartText** class also creates independent text objects. Other classes that display text also use it internally.

ChartTitle This class displays a text string as the title or footer of the chart.

AxisTitle	This class displays a text string as the title for an axis. The axis title position is outside of the axis label area. Axis titles for y-axes are rotated 90 degrees.
ChartLabel	This class is the abstract base class of labels that require special formatting.
NumericLabel	This class is a concrete implementation of the **ChartLabel** class and it displays formatted numeric values.
TimeLabel	This class is a concrete implementation of the **ChartLabel** class and it displays formatted **GregorianCalendar** dates.
ElapsedTimeLabel	This class is a concrete implementation of the **ChartLabel** class and it displays numeric values formatted as elapsed time strings **(12:32:21).**
StringLabel	This class is a concrete implementation of the **ChartLabel** class that formats string values for use as axis labels.

Miscellaneous Chart Classes

Marker
ChartImage
ChartShape
ChartSymbol

Various classes are used to position and draw objects that can be used as standalone objects in a graph, or as elements of other plot objects.

Marker	This class displays one of five marker types in a graph. The marker is used to create data cursors, or to mark data points.
ChartImage	This class encapsulates a **Java Image** class, defining a rectangle in chart coordinates that the image is placed in. JPEG and other image files can be imported using the Java **Image** class and displayed in a chart.
ChartShape	This class encapsulates a **Java Shape** class, placing the shape in a chart using a position defined in chart

coordinates. A chart can display any object that can be defined using **Shape** class.

ChartSymbol This class defines symbols used by the **SimplePlot** scatter plot functions. Pre-defined symbols include square, triangle, diamond, cross, plus, star, line, horizontal bar, vertical bar, 3D bar and circle.

Mouse Interaction Classes

ChartMouseListener
- **MoveObj**
- **FindObj**
- **DataToolTip**
  - **MoveData**
- **MagniView**
- **MoveCoordinates**
- **ChartZoom**

Marker
- **DataCursor**
  - **MoveData**

Several classes implement Swing **MouseInputListener** interface class. The **ChartMouseListener** class implements a generic interface for managing mouse events in a graph window. The **DataCursor**, **MoveData**, **MoveObj** and **ChartZoom** classes also implement the **MouseInputListener** interface, using the mouse to mark, move and zoom chart objects and data.

ChartMouseListener This class implements the Java **javax.swing.event.MouseInputListener** interface. It traps generic mouse events (button events and mouse motion events) that take place in a **ChartView** window. A programmer can derive a class from **ChartMouseListener** and override the methods for mouse events, creating a custom version of the class.

MoveObj This class extends the **ChartMouseListener** class and it can select chart objects and move them. Moveable chart objects include axes, axes labels, titles, legends, arbitrary text, shapes and images. Use the **MoveData** class to move objects derived from **SimplePlot**.

FindObj This class extends the **ChartMouseListener** class, providing additional methods that selectively determine what graphical objects intersect the mouse cursor.

DataCursor This class combines the **MouseInputListener** interface and **Marker** class. Press a mouse button and the selected data cursor (horizontal and/or vertical line, cross hairs, or a small box) appears at the point of the mouse cursor. The data cursor tracks the mouse motion as long as the mouse button is pressed. Release the button and the data cursor disappears. This makes it easier to line up the mouse position with the tick marks of an axis.

MoveData This class selects and moves individual data points of an object derived from the **SimplePlot** class.

DataToolTip A data tooltip is a popup box that displays the value of a data point in a chart. The data value can consist of the x-value, the y-value, x- and y-values, group values and open-high-low-close values, for a given point in a chart.

ChartZoom This class implements mouse controlled zooming for one or more simultaneous axes. The user starts zooming by holding down a mouse button with the mouse cursor in the plot area of a graph. The mouse is dragged and then released. The rectangle established by mouse start and stop points defines the new, zoomed, scale of the associated axes. Zooming has many different modes. Some of the combinations are:

- One x or one y axis
- One x and one y axes
- One x and multiple y axes
- One y and multiple x axes
- Multiple x and y axes

MagniView This class implements mouse controlled magnification for one or more simultaneous axes. This class implements a chart magnify class based on the **MouseListener** class. It uses two charts; the source chart and the target chart. The source chart displays the chart in its unmagnified state. The target chart displays the chart in the magnified state. The mouse positions a **MagniView** rectangle within the source chart, and the target chart is re-scaled and redrawn to match the extents of the **MagniView** rectangle from the source chart.

MoveCoordinates This class extends the **MouseListener** class and it can move the coordinate system of the underlying chart, analogous to moving (chaging the coordinates of) an internet map by “grabbing” it with the mouse and dragging it.

File and Printer Rendering Classes

ChartPrint
ChartBufferedImage

ChartPrint This class implements the Java **Printable** interface. It can select, setup, and output a chart to a printer.

ChartBufferedImage This class will convert an **ChartView** object to a Java **BufferedImage** object. Optionally, the class saves the buffered image to a JPEG file.

Miscellaneous Utility Classes

ChartConstants
ChartCalendar
CSV
ChartDimension
ChartPoint2D
GroupPoint2D
DoubleArray
DoubleArray2D
BoolArray
ChartPoint3D
NearestPointData
TickMark
Polysurface
ChartRectangle2D

ChartConstants This interface consolidates all of the constant values used in the com.quinncurtis.chart2djava package**.**

ChartCalendar This class contains utility routines used to process **GregorianCalendar** date objects.

CSV	This is a utility class for reading and writing CSV (Comma Separated Values) files.
ChartDimension	This is a utility class for handling dimension (height and width) information using doubles, rather than the integers used by the Size class.
ChartPoint2D	This class encapsulates an xy pair of values as doubles.
GroupPoint2D	This class encapsulates an x-value, and an array of y-values, representing the x and y values of one column of a group data set.
DoubleArray	This class is used as an alternative to the standard Array class, adding routines for resizing of the array, and the insertion and deletion of double based data elements.
DoubleArray2D	This class is used as an alternative to the standard Array class, adding routines for resizing of the array, and the insertion and deletion of double based data elements.
BoolArray	This class is used as an alternative to the standard Java Array class, adding routines for resizing of the array, and the insertion and deletion of boolean based data elements.
ChartPoint3D	This class encapsulates an xyz set of double values used to specify 3D data values.
NearestPointData	This is a utility class for returning data that results from nearest point calculations.
TickMark	The axis classes use this class to organize the location of the individual tick marks of an axis.
Polysurface	This is a utility class that defines complex 3D shapes as a list of simple 3-sided polygons. The contour plotting routines use it.
ChartRectangle2D	This is a utility class that extends the Rectangle2D.Double class.

A diagram depicts the class hierarchy of the QCChart2D for Java library.


ChartConstants
ChartObj
Arrow
GregorianCalendar
CSV
ChartDimension
ChartPoint3D
GroupPoint2D
NearestPointData
Polysurface
ChartScale
LinearScale
LogScale
TimeScale
ElapsedTimeScale
UserCoordinates
WorldCoordinates
WorkingCoordinates
PhysicalCoordinates
CartesianCoordinates
PolarCoordinates
AntennaCoordinates
TimeCoordinates
ChartDataset
SimpleDataset
TimeSimpleDataset
ElapsedTimeSimpleDataset
ContourDataset
GroupDataset
TimeGroupDataset
ElapsedTimeGroupDataset
AutoScale
LinearAutoScale
LogAutoScale
TimeAutoScale
ElapsedTimeAutoScale
MouseListener
MoveObj
FindObj
DataToolTip
ChartZoom
MagniView
MoveCoordinates
DataCursor
MoveData
ChartAttribute
ChartPrint
ChartBufferedImage
javax.swing.JPanel
ChartView
Rectangle2D.Double
ChartRectangle2D
Point2D.Double
ChartPoint2D
DoubleArray
DoubleArray2D
BoolArray
Polysurface
GraphObj
AntennaAnnotation
TickMark
Axis
LinearAxis
PolarAxes
AntennaAxes
LogAxis
TimeAxis
ChartText
ChartTitle
AxisTitle
ChartLabel
NumericLabel
BarDatapointValue
TimeLabel
ElapsedTimeLabel
StringLabel
AxisLabels
NumericAxisLabels
TimeAxisLabels
ElapsedTimeAxisLabels
StringAxisLabels
PolarAxesLabels
AntennaAxesLabels
Grid
PolarGrid
AntennaGrid
LegendItem
BubblePlotLegendItem
Legend
StandardLegend
BubblePlotLegend
ChartPlot
SimplePlot
SimpleLinePlot
SimpleBarPlot
SimpleScatterPlot
SimpleLineMarkerPlot
SimpleVersaPlot
GroupPlot
ArrowPlot
BubblePlot
CandlestickPlot
CellPlot
ErrorBarPlot
FloatingBarPlot
FloatingStackedBarPlot
GroupBarPlot
HistogramPlot
LineGapPlot
MultiLinePlot
OHLCPlot
StackedBarPlot
StackedLinePlot
BoxWhiskerPlot
GroupVersaPlot
PieChart
PolarPlot
PolarLinePlot
PolarScatterPlot
AntennaPlot
AntennaLinePlot
AntennaScatterPlot
AntennaLineMarkerPlot
Background
ChartImage
ChartShape
ChartSymbol
Marker

ChartZoom

4. Process Variable and Alarm Classes

RTProcessVar
RTAlarm
RTAlarmEventArgs

The **RTProcessVar** class is the core data storage object for all of the real-time indicator classes. The **RTProcessVar** class represents a single process variable, complete with limit values, an unlimited number of high and low alarms, historical data storage, and descriptive strings for use in displays.

Indicators that display the current value of a single process variable, the **RTBarIndicator**, the **RTMeterIndicator** and **RTPanelMeter** classes for example, reference back to a single **RTProcessVar** object. Indicators that display the current values of multiple process variables, the **RTMultiBarIndicator**, **RTMultiValueAnnunciator** and **RTFormControlGrid** classes, reference back to a collection of **RTProcessVar** objects. Even though an **RTProcessVar** object is in a multi-value indicator collection with other **RTProcessVar** objects, it maintains its own unique settings for limit values, alarm limits and descriptive strings.

The **RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot** classes provide a link between the **RTProcessVar** class and the charting routines in the **QCChart2D** charting package. The **RTSimpleSingleValuePlot** class combines any of the **QCChart2D SimplePlot** classes with an **RTProcessVar** object, and the **RTGroupMultiValuePlot** class combines any of the **QCChart2D GroupPlot** classes with a collection of **RTProcessVar** objects. The **RTProcessVar** class manages a historical data buffer based on the **QCChart2D ChartDataset** class. Each time the current value of the **RTProcessVar** object is updated, it is time-stamped and its value appended to the internal **ChartDataset**. The time stamp can either be explicitly supplied in the update call, or it can be automatically derived from the system clock. From there it can be plotted in static or scrolling plots.

The **RTProcessVar** class contains a collection of **RTAlarm** objects. Each alarm object represents a unique alarm condition: either a greater than alarm or a less than alarm, based on the specified limit value. The **RTAlarm** class also specifies alarm text strings, alarm colors, and the alarm hysteresis value. An **RTProcessVar** object can hold an unlimited number of **RTAlarm** objects. Every time an **RTProcessVar** object is updated with new values, every alarm is checked and an alarm event is generated if the alarm conditions are met. The programmer can hook into the alarm events using alarm event delegates.

Real-Time Process Variable

Class RTProcessVar

ChartObj
|
+-- RTProcessVar

Real-time data is stored in **RTProcessVar** classes. The **RTProcessVar** class is designed to represent a single process variable, complete with limit values, an unlimited number of high and low alarms, historical data storage, and descriptive strings for use in displays. It has two main constructors.

RTProcessVar constructors

```
public RTProcessVar(
   string tagname,
   ChartAttribute defaultattribute
);
public RTProcessVar(
   SimpleDataset dataset,
   ChartAttribute defaultattribute
);
```

Parameters

tagname
A string representing the tag name of the process variable.

dataset
A dataset that will be used to hold historical values for the process variable. If no tag name is supplied in the constructor the tag name for the process variable will be taken from the **ChartDataset.DataName** property of the dataset.

defaultattribute
Specifies the default attributes for the process variable.

Once created, the **RTProcessVar** object is updated using the **setCurrentValue** method. The method has several overloads:

```
public  void setCurrentValue(
   GregorianCalendar gv,
   double pv
);

public  void setCurrentValue(
   double timestamp,
   double pv
);
public  void setCurrentValue(
   double pv
);
```

Parameters

gv
The time stamp as a **GregorianCalendar** object for the process variable.

timestamp
The time stamp, in milliseconds, for the process variable.

pv
The value of the process variable.

If the time stamp value is not explicitly provided, as in the case of the last overloaded method, the current time as stored in the system clock is used as the time stamp.

Alarms are added to an **RTProcessVar** object using the **RTProcessVar.addAlarm or RTProcessVar.addCloneAlarm** methods. The **addCloneAlarm** method clones the passed in alarm object, so that the same **RTAlarmObject** can be used to initialize multiple **RTProcessVar** objects without a conflict occurring.

```
public void addAlarm(
   RTAlarm alarmobj
);
public RTAlarm addCloneAlarm(
   RTAlarm alarmobj
);
```

Parameters

alarmobj
A reference to the **RTAlarm** object that is to be added to the process variables alarm list.

Configure the **RTAlarm** object and then add it to the **RTProcessVar** object. If you plan to use the **RTAlarm** event handlers, make sure that you create a unique **RTAlarm** object for every alarm added to a **RTProcessVar** object.

The most commonly used **RTProcessVar** properties are:

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAlarmStateEventEnable**(value) and value := **getAlarmStateEventEnable**();

AlarmStateEventEnable	Get/Set the flag for the alarm state event enable. Set to true to enable alarm checking.
AlarmTransitionEventEnable	Get/Set the flag for the

	AlarmTransitionEventHandler delegate enable. Set to true to enable the **AlarmTransitionEventHandler** delegate.
CurrentValue	Get the process variable current value.
DatasetEnableUpdate	Get/Set to true to enable historical data collection in the process variable dataset.
DefaultAttribute	Get/Set the default attributes for the process variable.
DefaultMaximumDisplayValue	Get/Set maximum allowable display value for the process variable.
DefaultMinimumDisplayValue	Get/Set minimum allowable display value for the process variable.
DetailedDescription	Get/Set the process variable detailed description string.
GoodValue	Get/Set set to false designates that the current value is a bad value.
MaximumValue	Get/Set maximum allowable value for the process variable.
MinimumValue	Get/Set minimum allowable value for the process variable.
PrevCurrentValue	Get the process variable previous current value.
PrevTimeStamp	Get the process variable previous time stamp value.
ProcessVarDataset	Get/Set the process variable dataset.
ShortDescription	Get/Set the process variable short description string.
TagName	Get/Set the process variable tag string.
TimeStamp	Get the process variable time stamp value.
UniqueIdentifier	Get/Set the process variable unique identifier string.
UnitsString	Get/Set the process variable units string.

Public Instance Events

AlarmStateEventHandler	Delegate for notification each time the check of a process variable produces an alarm state condition.
AlarmTransitionEventHandler	Delegate for notification each time the check of a process variable produces a change of state in alarm state condition.

A complete listing of **RTProcessVar** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example of Creating an RTProcessVar Object

The example below creates and updates an **RTProcessVar** object that does not use alarms. The example was extracted from the Treadmill example program, method **InitializeGraph**. See the example in the **RTAlarm** section of the manual for one that uses alarms.

```
double  METSValue = 0;
RTProcessVar METS;
.
.
.
ChartAttribute defaultattrib = new ChartAttribute(Color.green, 1.0,
ChartConstants.LS_SOLID, Color.green);
METSValue = 0;
METS = new RTProcessVar("METS", defaultattrib);
METS.setShortDescription("Metabolic Equivalents");
METS.setMinimumValue(0);
METS.setMaximumValue(100);
METS.setDefaultMinimumDisplayValue(0);
METS.setDefaultMaximumDisplayValue(100);
METS.setDatasetEnableUpdate(true); // Make sure this is on for scrolling graphs
METS.setCurrentValue( METSValue);
```

Real-Time Alarms

Class RTAlarm

ChartObj
|
+-- RTAlarm

The **RTAlarm** class stores alarm information for the **RTProcessVar** class. The **RTAlarm** class specifies the type of the alarm, the alarm color, alarm text messages and alarm hysteresis value. The **RTProcessVar** classes can hold an unlimited number of **RTAlarm** objects in an internal **Vector**.

RTProcessVar constructors

```
public RTAlarm(
   RTProcessVar processvar,
   int alarmtype
);
```

```
public RTAlarm(
   int alarmtype,
   double alarmlimitvalue
);

public RTAlarm(
   RTProcessVar processvar,
   int alarmtype,
   double alarmlimitvalue
);

public RTAlarm(
   RTProcessVar processvar,
   int alarmtype,
   double alarmlimitvalue,
   string normalmessage,
   string alarmmessage
);

public RTAlarm(
   RTProcessVar processvar,
   int alarmtype,
   double alarmlimitvalue,
   string normalmessage,
   string alarmmessage,
   double hysteresisvalue
);
```

Parameters

processvar

Specifies the process variable that the alarm is for. When you add an **RTAlarm** object to an **RTProcessVar** object, this field will be overwritten with a reference to the **RTProcessVar** object that was added to.

alarmtype

Specifies the alarm type: RT_ALARM_NONE, RT_ALARM_LOWERTHAN, or RT_ALARM_GREATERTHAN.

alarmlimitvalue

Specifies the alarm limit value.

normalmessage

Specifies the message displayed when there is no alarm.

alarmmessage

Specifies the alarm message.

hysteresisvalue

Specifies the hysteresis value of the alarm. This is used to prevent alarms from toggling between states due to noise in the system when the process variable is very close to an alarm threshold. After an alarm has been triggered, the process variable must cross the alarm threshold in the opposite direction by the hysteresis value before it falls out of alarm. For example, if an RT_ALARM_GREATERTHAN alarm threshold is 70, then the process value will always go into alarm once the threshold value of 70 is exceeded. If the hysteresis value is 2, then the process variable will not fall out of alarm until the process value is less than (alarmlimitvalue – hysteresisvalue) = (70 – 2) = 68. If you don't want hysteresis set it equal to 0.0.

The most commonly used **RTAlarm** properties are:

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAlarmLimitValue**(value) and value := **getAlarmLimitValue**();

AlarmLimitValue	Get/Set the alarm limit value.
AlarmMessage	Get/Set the current alarm message.
AlarmState	Get/Set the alarm state, true if the last call to **CheckAlarm** show that the process variable currently in alarm.
AlarmSymbolColor	Get/Set the alarm symbol color.
AlarmTextColor	Get/Set the alarm text color.
AlarmType	Get/Set the alarm type: RT_ALARM_NONE, RT_ALARM_LOWERTHAN, or RT_ALARM_GREATERTHAN.
HysteresisValue	Get/Set the alarm hysteresis value.

A complete listing of **RTAlarm** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Once an **RTAlarm** object is added to an **RTProcessVar** object, alarm checking takes place every time the **RTProcessVar.setCurrentValue** method is called. Any displays dependent on the alarm will not change until the **ChartView.updateDraw** method is called, forcing a repaint of the view.

Example of RTAlarm objects added to an RTProcessVar object

The example below creates and updates an **RTProcessVar** object that uses alarms. It does not however generate alarm events that notify user-defined alarm handlers. The example was extracted from the Treadmill example program, method **InitializeGraph**. See the example in the **RTAlarmEventArgs** section for one that generates alarm events.

```
ChartAttribute defaultattrib = new ChartAttribute(Color.green, 1.0,
ChartConstants.LS_SOLID, Color.green);

RTAlarm lowheartratealarm = new RTAlarm(ChartConstants.RT_ALARM_LOWERTHAN, 30);
lowheartratealarm.setAlarmMessage("Low Heart Rate");
```

```
lowheartratealarm.setAlarmSymbolColor(Color.blue);
lowheartratealarm.setAlarmTextColor(Color.blue);
RTAlarm highheartratealarm = new RTAlarm(ChartConstants.RT_ALARM_GREATERTHAN,
160);
highheartratealarm.setAlarmMessage("High Heart Rate");
highheartratealarm.setAlarmSymbolColor(Color.red);
highheartratealarm.setAlarmTextColor(Color.red);

heartRate = new RTProcessVar("Heart Rate", defaultattrib);
heartRate.setMinimumValue(0);
heartRate.setMaximumValue(300);
heartRate.setDefaultMinimumDisplayValue(0);
heartRate.setDefaultMaximumDisplayValue(200);
heartRate.addAlarm(lowheartratealarm);
heartRate.addAlarm(highheartratealarm);
//  heartRate.setalarmTransitionEventEnable (true);
//  heartRate.addAlarmTransitionEventListener(this);
heartRate.setCurrentValue( heartRateValue);
```

Real-Time Alarms Event Handling

Class RTAlarmEventArgs

ChartObj
|
+-- RTAlarmEventArgs

The **RTProcessVar** class can throw an alarm event based on either the current alarm state, or an alarm transition from one alarm state to another. The **RTAlarmEventArgs** class is used to pass alarm data to the event handler. If you want the alarm event to be called only on the initial transition from the no-alarm state to the alarm state, set the **RTProcessVar AlarmTransitionEventEnable** to true and the **RTProcessVar AlarmStateEventEnable** to false**.** In this case you will get one event when the process variable goes into alarm, and one when it comes out of alarm. If you want a continuous stream of alarm events, as long as the **RTAlarm** object is in alarm, set the **RTProcessVar AlarmTransitionEventEnable** to false and the **RTProcessVar AlarmStateEventEnable** to true. The alarm events will be generated at the same rate as the **RTProcessVar.setCurrentValue()** method is called,

The alarm event handler is defined in the RTAlarmEventListener Interface as the method **AlarmEventChanged.** Implement the **RTAlarmEventListener** Interface in the class you want to process the alarm event. Implementing the **RTAlarmEventListener** Interface consists of adding implements **RTAlarmEventListener** in the class declaration and adding the **AlarmEventChanged** method to the body of the class implementation.

```
public class Treadmill extends ChartView implements  RTAlarmEventListener {
    .
.
.
public void AlarmEventChanged(RTProcessVar sender, RTAlarmEventArgs e)
{  RTAlarm alarm = e.getEventAlarm();
   double alarmlimitvalue = alarm.getAlarmLimitValue();
   String alarmlimitstring = Double.toString(alarmlimitvalue);

   if (alarm.getAlarmState())
     System.out.println("Warning - Heart Rate Alarm Level " + alarmlimitstring +
" Exceeded");
   else
     System.out.println("Heart Rate transitioned back to normal range.");
}
```

RTAlarmEventArgs constructors

You don't really need the constructors since **RTAlarmEventArgs** objects are created inside the **RTProcessVar** class when an alarm event needs to be generated. Here they are anyway.

```
public RTAlarmEventArgs(
   RTProcessVar pv,
   RTAlarm alarm,
   int channel
);
```

Parameters

pv

The **RTProcessVar** object associated with the alarm event.

alarm

The **RTAlarm** object associated with the alarm event.

channel

The channel number associated with the alarm event.

The most commonly used **RTAlarmEventArgs** properties are:

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAlarmChannel**(value) and value := **getAlarmChannel**();

AlarmChannel	Get/Set the alarm channel object.
EventAlarm	Get/Set the **RTAlarm** object.associated with the alarm
ProcessVar	Get/Set the **RTProcessVar** object. associated with the alarm

A complete listing of **RTAlarmEventArgs** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example

Setup and enable an alarm transition event handler in the following manner:

```
heartRate.setAlarmTransitionEventEnable (true);
heartRate.addAlarmTransitionEventListener(this);
```

where the handler method is **this. AlarmEventChanged**, as defined by the **RTAlarmEventListener** Interface implemented by the Treadmill class.

```
public void AlarmEventChanged(RTProcessVar sender, RTAlarmEventArgs e)
{  RTAlarm alarm = e.getEventAlarm();
   double alarmlimitvalue = alarm.getAlarmLimitValue();
   String alarmlimitstring = Double.toString(alarmlimitvalue);

   if (alarm.getAlarmState())
     System.out.println("Warning - Heart Rate Alarm Level " + alarmlimitstring +
" Exceeded");
   else
     System.out.println("Heart Rate transitioned back to normal range.");
}
```

Setup and enable an alarm state event handler in an identical manner:

```
heartRate.setAlarmStateEventEnable (true);

heartRate.addAlarmStateEventListener(this);
```

where the handler method would also be the **RTAlarmEventListener** Interface implemented by the Treadmill class.

Example of RTProcessVar, RTAlarm and Alarm Event Handlers

The example below creates and updates an RTProcessVar object that uses alarms and a user-defined alarm event handler. The example was extracted from the Treadmill example program, method InitializeGraph. See the example in the RTAlarmEventArgs section for one that generates alarm events.

```
ChartAttribute defaultattrib = new ChartAttribute(Color.green, 1.0,
ChartConstants.LS_SOLID, Color.green);

RTAlarm lowheartratealarm = new RTAlarm(ChartConstants.RT_ALARM_LOWERTHAN, 30);
```

```
lowheartratealarm.setAlarmMessage("Low Heart Rate");
lowheartratealarm.setAlarmSymbolColor(Color.blue);
lowheartratealarm.setAlarmTextColor(Color.blue);
RTAlarm highheartratealarm = new RTAlarm(ChartConstants.RT_ALARM_GREATERTHAN,
160);
highheartratealarm.setAlarmMessage("High Heart Rate");
highheartratealarm.setAlarmSymbolColor(Color.red);
highheartratealarm.setAlarmTextColor(Color.red);

heartRate = new RTProcessVar("Heart Rate", defaultattrib);
heartRate.setMinimumValue(0);
heartRate.setMaximumValue(300);
heartRate.setDefaultMinimumDisplayValue(0);
heartRate.setDefaultMaximumDisplayValue(200);
heartRate.addAlarm(lowheartratealarm);
heartRate.addAlarm(highheartratealarm);
heartRate.setalarmTransitionEventEnable (true);
heartRate.addAlarmTransitionEventListener(this);
heartRate.setCurrentValue( heartRateValue);
.
.
.
public void AlarmEventChanged(RTProcessVar sender, RTAlarmEventArgs e)
{  RTAlarm alarm = e.getEventAlarm();
   double alarmlimitvalue = alarm.getAlarmLimitValue();
   String alarmlimitstring = Double.toString(alarmlimitvalue);

   if (alarm.getAlarmState())
     System.out.println("Warning - Heart Rate Alarm Level " + alarmlimitstring +
" Exceeded");
   else
     System.out.println("Heart Rate transitioned back to normal range.");
}
```

5. Panel Meter Classes

RTNumericPanelMeter
RTAlarmPanelMeter
RTStringPanelMeter
RTTimePanelMeter
RTElapsedTimePanelMeter
RTFormControlPanelMeter

The **RTPanelMeter** derived classes are special cases of the single value indicator classes that are used throughout the software to display real-time data in a text format. Panel meters are available for numeric values, string values, time/date values and alarm values. All of the panel meter classes have a great many options for controlling the text font, color, size, border and background of the panel meter rectangle. **RTPanelMeter** objects are used in two ways. First, they can be standalone, and once attached to an **RTProcessVar** object they can be added to a **ChartView** as any other **QCChart2D GraphObj** derived class. Second, they can be attached to most of the single channel and multiple channel indicators, such as **RTBarIndicator**, **RTMultiBarIndicator**, **RTMeterIndicator** and **RTAnnunciator** objects, where they provide text output in addition to the indicators graphical output.

Digital SF Font

In our example programs the panel meters often use a simple 7-segment display font. It gives the displays an anachronistic look typical of older, dedicated instruments. This font is not part of the Java installation, but it is included with our software. It is a public domain font that has the family name Digital SF (with a space between “Digital” and “SF”. It resides in the Digital.TTF file found in the Quinn-Curtis\lib subdirectory. In order to use the font in Java programs you should copy this file to the Windows\Fonts subdirectory AND reboot your computer. The reboot forces Windows to re-enumerate the fonts. Once that is done, it can be used like any other TrueType font. All of our example programs assume that the font has been copied to the Windows\Fonts subdirectory and the computer rebooted. The example below creates an instance of the Digial SF font.

```
Font font12Numeric = new Font("Digital SF",Font.PLAIN, 12);
```

The font can be downloaded from the link:
http://www.webfontlist.com/pages/station.asp?ID=10643&x=Fonts if you want to download your own copy.

Panel Meters

Class RTPanelMeter

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter

The **RTPanelMeter** is an abstract class for the other panel meter classes. While it cannot be instantiated, it does contain properties and methods common to all panel meters. A summary of these properties is listed below.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAlarmIndicatorColorMode**(value) and value := **getAlarmIndicatorColorMode**();

AlarmIndicatorColorMode ***(inherited from RTSingleValueIndicator)***	Get/Set whether the color of the indicator objects changes on an alarm. Use one of the constants: RT_INDICATOR_COLOR_NO_ALARM_CHANGE, RT_INDICATOR_COLOR_CHANGE_ON_ALARM.
ContrastTextAlarmColor (inherited from **RTPanelMeter**)	Set/Get a contrast color to use for text when the object is in alarm, and the background color of the panel meter changes.
CurrentProcessValue (inherited from **RTSingleValueIndicator**)	Get the current process value of the primary channel.
Frame3DEnable (inherited from **RTPanelMeter**)	Set/Get to true to enable a 3D frame for the panel meter.
PanelMeterNudge (inherited from **RTPanelMeter**)	Set/Get the xy values of the **PanelMeterNudge** property. The **PanelMeterNudge** property moves the relative position, using window device coordinates, of the text relative to the specified location of the text.
PanelMeterPosition (inherited from **RTPanelMeter**)	Set/Get the panel meter position value. Use one of the panel meter position constants. See table for positioning constants.
PositionReference (inherited from **RTPanelMeter**)	Set/Get an **RTPanelMeter** object used as a positioning reference for this **RTPanelMeter** object.
PrimaryChannel (inherited from **RTPlot**)	Set/Get the primary channel of the indicator.
RTDataSource (inherited from **RTSingleValueIndicator**)	Get/Set the array list holding the **RTProcessVar** variables for the indicator.
RTPlotObj (inherited from	Set/Get the reference **RTPlot** object.

RTPanelMeter)	
TextColor	Set/Get the text color of the panel meter.

A complete listing of **RTPanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Positioning Panel Meters

The most complicated thing about panel meters is getting them positioned where you want. There are over 30 positioning constants that can be used to position panel meters with respect to graph objects, the plot area, and graph area of the associated graph. In addition to the positioning constants, you can explicitly place the panel meter anywhere that you want in a graph using the CUSTOM_POSITION position constant in conjunction with the **RTPanelMeter.setLocation** method. The table below summarizes the panel meter positioning constants used in the software.

Positioning Constants

CUSTOM_POSITION	Custom position specified using the RTPanelMeter.setLocation method. Position can be set using the DEV_POS, PHYS_POS, NORM_GRAPH_POS, NORM_PLOT_POS coordinate systems. Set the justification of the panel meter box using the StringLabel or NumericLabel template of the specific panel meter class.
CENTERED_BAR	Used when the panel meter is attached to a bar indicator. Centers the panel meter inside the bar indicator. If the object is not a bar indicator the panel meter is centered inside the plotting area. Text justification is set to (JUSTIFY_CENTER, JUSTIFY_CENTER)
OUTSIDE_BAR	Used when the panel meter is attached to a bar indicator. Places the panel meter on the outside edge of the bar indicator. If the object is not a bar indicator the panel meter is placed on the outside edge of the plotting area maximum. Text justification depends on the bar orientation: Vertical Bars (JUSTIFY_CENTER, JUSTIFY_MIN), Horizontal Bars (JUSTIFY_MIN, JUSTIFY_CENTER).
INSIDE_BAR	Used when the panel meter is attached to a bar indicator. Places the panel meter on the inside edge of the bar indicator. If the object is not a bar indicator the panel meter is placed on the inside edge of the plotting area maximum. Text justification depends on the bar orientation: Vertical Bars (JUSTIFY_CENTER, JUSTIFY_MAX), Horizontal Bars (JUSTIFY_MAX, JUSTIFY_CENTER).

INSIDE_BARBASE	Used when the panel meter is attached to a bar indicator. Places the panel meter on the inside edge of the bar base of the indicator. If the object is not a bar indicator the panel meter is placed on the inside edge of the plotting area minimum. Text justification depends on the bar orientation: Vertical Bars (JUSTIFY_CENTER, JUSTIFY_MIN), Horizontal Bars (JUSTIFY_MIN, JUSTIFY_CENTER).
OUTSIDE_BARBASE	Used when the panel meter is attached to a bar indicator. Places the panel meter on the outside edge of the bar base of the indicator. If the object is not a bar indicator the panel meter is placed on the outside edge of the plotting area maximum. Text justification depends on the bar orientation: : Vertical Bars (JUSTIFY_CENTER, JUSTIFY_MAX), Horizontal Bars (JUSTIFY_MAX, JUSTIFY_CENTER).
INSIDE_INDICATOR	Same as INSIDE_BAR.
OUTSIDE_INDICATOR	Same as OUTSIDE_BAR.
BELOW_REFERENCED_TEXT	Used when it is desired that the panel meter be positioned next to another object. Places the panel meter below the reference object. Specify the position reference object using the **PanelMeter.setPositionReference** method. Text justification is set to (JUSTIFY_CENTER, JUSTIFY_MAX).
ABOVE_REFERENCED_TEXT	Used when it is desired that the panel meter be positioned next to another object. Places the panel meter above the reference object. Specify the position reference object using the **PanelMeter.setPositionReference** method. Text justification is set to (JUSTIFY_CENTER, JUSTIFY_MIN).
RIGHT_REFERENCED_TEXT	Used when it is desired that the panel meter be positioned next to another object. Places the panel meter to the right of the reference object. Specify the position reference object using the **PanelMeter.setPositionReference** method. Text justification is set to (JUSTIFY_MIN, JUSTIFY_CENTER).
LEFT_REFERENCED_TEXT	Used when it is desired that the panel meter be positioned next to another object. Places the panel meter to the left of the reference object. Specify the position reference object using the **PanelMeter.setPositionReference** method. Text justification is set to (JUSTIFY_MAX, JUSTIFY_CENTER).
BELOW_CENTERED_PLOTAREA	Positions the panel meter centered, below the plot area. Text justification is set to (JUSTIFY_CENTER, JUSTIFY_MAX).

ABOVE_CENTERED_PLOTAREA	Positions the panel meter centered, above the plot area. Text justification is set to (JUSTIFY_CENTER, JUSTIFY_MIN).
LEFT_CENTERED_PLOTAREA	Positions the panel meter centered, to the left of the plot area. Text justification is set to (JUSTIFY_MAX, JUSTIFY_CENTER).
RIGHT_CENTERED_PLOTAREA	Positions the panel meter centered, to the right of the plot area. Text justification is set to (JUSTIFY_MIN, JUSTIFY_CENTER).
GRAPHAREA_TOP	Positions the panel meter centered, at the top edge of the graph area. Text justification is set to (JUSTIFY_CENTER, JUSTIFY_MAX).
GRAPHAREA_BOTTOM.	Positions the panel meter centered, at the bottom edge of the graph area. Text justification is set to (JUSTIFY_CENTER, JUSTIFY_MIN).
RADIUS_BOTTOM	Used when the panel meter is attached to a meter indicator. Places the panel meter at the bottom, at the radius of the **MeterCoordinates** system. You can set the text justification however you want.
RADIUS_TOP	Used when the panel meter is attached to a meter indicator. Places the panel meter at the top, at the radius of the **MeterCoordinates** system. You can set the text justification however you want.
RADIUS_LEFT	Used when the panel meter is attached to a meter indicator. Places the panel meter on the left, at the radius of the **MeterCoordinates** system. You can set the text justification however you want.
RADIUS_RIGHT	Used when the panel meter is attached to a meter indicator. Places the panel meter on the right, at the radius of the **MeterCoordinates** system. You can set the text justification however you want.
RADIUS_CENTER	Used when the panel meter is attached to a meter indicator. Places the panel meter at the center of the radius of the **MeterCoordinates** system. You can set the text justification however you want.
OUTSIDE_RADIUS_BOTTOM	Used when the panel meter is attached to a meter indicator. Places the panel meter at the bottom, centered on the outside edge of the radius of the **MeterCoordinates** system. Text

	justification is (JUSTIFY_CENTER, JUSTIFY_MAX).
INSIDE_RADIUS_BOTTOM	Used when the panel meter is attached to a meter indicator. Places the panel meter at the bottom, centered on the inside edge of the radius of the **MeterCoordinates** system. Text justification is (JUSTIFY_CENTER, JUSTIFY_MIN).
CENTER_RADIUS_BOTTOM	Used when the panel meter is attached to a meter indicator. Places the panel meter at the bottom, centered on the inside edge of the radius of the **MeterCoordinates** system. Text justification is (JUSTIFY_CENTER, JUSTIFY_CENTER).
OUTSIDE_RADIUS_TOP	Used when the panel meter is attached to a meter indicator. Places the panel meter at the top, centered on the outside edge of the radius of the **MeterCoordinates** system. Text justification is (JUSTIFY_CENTER, JUSTIFY_MIN).
INSIDE_RADIUS_TOP	Used when the panel meter is attached to a meter indicator. Places the panel meter at the top, centered on the inside edge of the radius of the **MeterCoordinates** system. Text justification is (JUSTIFY_CENTER, JUSTIFY_MAX).
CENTER_RADIUS_TOP	Used when the panel meter is attached to a meter indicator. Places the panel meter at the top, centered on the inside edge of the radius of the **MeterCoordinates** system. Text justification is (JUSTIFY_CENTER, JUSTIFY_CENTER).

A particularly useful property is **RTPanelMeter PanelMeterNudge**. After you get the panel meter positioned approximately where you want, you may find that it just a couple of pixels too close to some other object, whether it be an indicator, axis, or text object. You can nudge the panel meter in any direction with respect to its calculated position. The **PanelMeterNudge** property uses device (or pixel) coordinates.

```
RTNumericPanelMeter panelmeter =
    new RTNumericPanelMeter(pTransform1, panelmeterattrib);
panelmeter.setPanelMeterPosition (ChartConstants.OUTSIDE_PLOTAREA_MIN);
panelmeter.setPanelMeterNudge(0,4);
```

Numeric Panel Meter

Class RTNumericPanelMeter

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTNumericPanelMeter

The **RTNumericPanelMeter** class displays the floating point numeric value of an **RTProcessVar** object. It contains a template based on the **QCChart2D NumericLabel** class that is used to specify the font and numeric format information associated with the panel meter.

RTNumericPanelMeter constructors

```
public RTNumericPanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   ChartAttribute attrib
);

public RTNumericPanelMeter(
   PhysicalCoordinates transform,
   ChartAttribute attrib
);
```

Parameters

transform
The coordinate system for the new **RTNumericPanelMeter** object.
datasource
The process variable associated with the panel meter.
attrib
The color attributes of the panel meter indicator.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setNumericTemplate**(value) and value := **getNumericTemplate**();

NumericTemplate	Set/Get the **NumericLabel** template for the panel meter numeric value. The text properties associated with the panel meter are set using this property. In addition, the format of the numeric value and the number of digits to the right of the decimal point is also set here.

A complete listing of **RTNumericPanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example

The panel meter below, extracted from the Treadmill example, method **InitializeRightPanelMeters**, is an independent panel meter that displays the METSCumlative process variable. The positioning is accomplished using the **RTPanelMeter.setLocation** method**,** which in this case places the panel meter using normalized graph coordinates. The size of the panel meter is determined by two things, the font size, which in this case is 64, and the number of digits used in the display. In order to keep the panel meter box from constantly changing size if the number of digits changes, the panel meter box is sized to accommodate the range of values specified by the **RTProcessVar DefaultMinimumDisplayValue** and **RTProcessVar DefaultMaximumDisplayValue** properties.

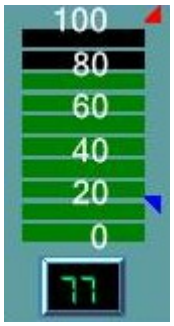

```
public void InitializeRightPanelMeters()
{
    Font numericfont = font64Numeric;
    Font trackbarTitlefont = font12Bold;

    ChartView chartVu  = this;
    CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0,
1.0);
    pTransform1.setGraphBorderDiagonal(0.0, 0.0, 1.0, 1.0) ;
    ChartAttribute attrib1 = new ChartAttribute (lightBlue,
3,ChartConstants.LS_SOLID, lightBlue);
    ChartAttribute panelmeterattrib = new
ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
    RTNumericPanelMeter panelmeter1 = new RTNumericPanelMeter(pTransform1,
METSCumlative,panelmeterattrib);
    panelmeter1.getNumericTemplate().setTextFont(numericfont);
    panelmeter1.getNumericTemplate().setDecimalPos(0);
    panelmeter1.setPanelMeterPosition(ChartConstants.CUSTOM_POSITION);
    panelmeter1.setLocation(0.81,0.3, ChartConstants.NORM_GRAPH_POS);
    chartVu.addChartObject(panelmeter1);
.
.
.
}
```

Example of RTNumericPanelMeter used with an RTBarIndicator

The panel meter below, extracted from the HybridCar example, method **InitializeBatteryChargeGraph**, is an **RTNumericPanelMeter** attached to a **RTBarIndicator** bar plot object.. The position of the panel meter is

OUTSIDE_PLOTAREA_MIN, which places underneath the dynamic bar. The size of the panel meter is determined by two things, the font size, which in this case is 14, and the number of digits used in the display.

Note: Unlike the previous example, the panel meter is not added to the **ChartView** (*chartVu* above); instead it is added to the **RTBarIndicator** (*barplot* below). The panel meter will assume the values of the **RTProcessVar** used by the bar plot.




```
RTBarIndicator barplot = new RTBarIndicator(pTransform1,
    batteryCharge,  barwidth,  barbase,
    attrib1,   barjust,  barorient);
.
.
.
barplot.setIndicatorSubType(ChartConstants.RT_BAR_SEGMENTED_SUBTYPE);


RTAlarmIndicator baralarms = new  RTAlarmIndicator(baraxis, barplot);
chartVu.addChartObject(baralarms);

ChartAttribute panelmeterattrib = new
ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
ChartAttribute paneltagmeterattrib = new
ChartAttribute(steelBlue,0,ChartConstants.LS_SOLID, Color.white);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(pTransform1,
panelmeterattrib);
panelmeter.setPanelMeterPosition(ChartConstants.OUTSIDE_PLOTAREA_MIN);
panelmeter.setTextColor(springGreen);
panelmeter.getNumericTemplate().setTextFont(font18Numeric);
panelmeter.getNumericTemplate().setDecimalPos(0);
panelmeter.setAlarmIndicatorColorMode(ChartConstants.RT_TEXT_BACKGROUND_COLOR_CHAN
GE_ON_ALARM);
panelmeter.setPanelMeterNudge(0,4);
barplot.addPanelMeter(panelmeter);
.
.
.
}
```

Alarm Panel Meter

Class RTAlarmPanelMeter

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTAlarmPanelMeter

The **RTAlarmPanelMeter** class displays the alarm state of an **RTProcessVar** object. It pulls alarm text and color information out of the associated **RTProcessVar** object. It contains a template based on the **QCChart2D StringLabel** class that is used to specify the font and numeric format information associated with the panel meter.

RTAlarmPanelMeter constructors

```
public RTAlarmPanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   ChartAttribute attrib
);
public RTAlarmPanelMeter(
   PhysicalCoordinates transform,
   ChartAttribute attrib
);
```

Parameters

transform
The coordinate system for the new **RTAlarmPanelMeter** object.
datasource
The process variable associated with the panel meter.
attrib
The color attributes of the panel meter indicator.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setNumericTemplate**(value) and value := **getNumericTemplate**();

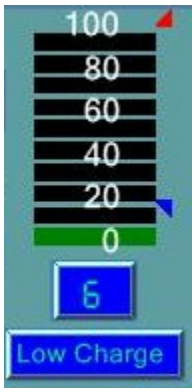
AlarmTemplate	Get/Set the string template defining the panel meter alarm string format. The text properties associated with the panel meter are set using this property.

A complete listing of **RTAlarmPanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example of RTAlarmPanelMeter used with RTBarIndicator

The panel meter below, extracted from the HybridCar example, method **InitializeBatteryChargeGraph,** adds an **RTAlarmPanelMeter** underneath the numeric panel meter.

Note: The **RTAlarmPanelMeter** uses the BELOW_REFERENCED_TEXT positioning constant, and sets the **RTAlarmPanelMeter.setPositionReference** to the numeric panel meter.




```
RTBarIndicator barplot = new RTBarIndicator(pTransform1,
    batteryCharge,  barwidth,  barbase,
    attrib1,   barjust,  barorient);
.
.
.
RTAlarmIndicator baralarms = new  RTAlarmIndicator(baraxis, barplot);
chartVu.addChartObject(baralarms);

ChartAttribute panelmeterattrib = new
ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
ChartAttribute paneltagmeterattrib = new
ChartAttribute(steelBlue,0,ChartConstants.LS_SOLID, Color.white);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(pTransform1,
panelmeterattrib);
.
.
.

barplot.addPanelMeter(panelmeter);

RTAlarmPanelMeter panelmeter2 = new RTAlarmPanelMeter(pTransform1,
panelmeterattrib);
panelmeter2.setPanelMeterPosition(ChartConstants.BELOW_REFERENCED_TEXT);
panelmeter2.setTextColor(springGreen);
panelmeter2.getAlarmTemplate().setTextFont(font10);
panelmeter2.setPositionReference( panelmeter);
panelmeter2.setAlarmIndicatorColorMode(ChartConstants.RT_TEXT_BACKGROUND_COLOR_CHA
NGE_ON_ALARM);
```

```
            barplot.addPanelMeter(panelmeter2);
.
.
```

String Panel Meter

Class RTStringPanelMeter

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTStringPanelMeter

The **RTStringPanelMeter** class displays a string, either an arbitrary string, or a string based on string data in the associated **RTProcessVar** object. It is usually used to display a channels tag string and units string, but it can also be used to display longer descriptive strings. It contains a template based on the **QCChart2D StringLabel** class that is used to specify the font and string format information associated with the panel.

RTStringPanelMeter constructors

```
public RTStringPanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   ChartAttribute attrib,
   int stringtype
);

public RTStringPanelMeter(
   PhysicalCoordinates transform,
   ChartAttribute attrib,
   int stringtype
);

public RTStringPanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   ChartAttribute attrib
);
```

Parameters

transform
The coordinate system for the new **RTStringPanelMeter** object.
datasource

The process variable associated with the panel meter.

attrib

The color attributes of the panel meter indicator.

stringtype

Specifies what string to display, whether it is one of the process variable strings, or a custom string. Use one of the Panel Meter string constants: RT_CUSTOM_STRING, RT_TAG_STRING , RT_UNITS_STRING. Specify a custom string and use the **StringTemplate.TextString** property to set the string.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setStringTemplate**(value) and value := **getStringTemplate**();

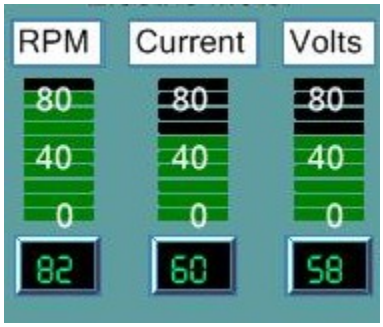
StringTemplate	Get/Set the string template defining the panel meter string format. The text properties associated with the panel meter are set using this property.

A complete listing of **RTStringPanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for RTStringPanelMeter used with RTMultiBarIndicator

The panel meter below, extracted from the HybridCar example, method **InitializeMotorVariablesGraph** , adds an **RTStringPanelMeter** above the motor variables multi-bar indicator. It is used to display the process variable tag name as the title for each bar in the multi-value indicator.

Note: The **RTStringPanelMeter** only needs to be added once to the **RTMultiBarIndicator**. It automatically picks up on the tag name for each **RTProcessVar** object referenced by the **RTMultiBarIndicator**.



```
RTMultiBarIndicator barplot = new RTMultiBarIndicator(pTransform1,
    motorvars,  barwidth, barspace, barbase,
    attribarray,  barjust,  barorient);
.
.
.

ChartAttribute panelmeterattrib = new
ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
ChartAttribute paneltagmeterattrib = new
ChartAttribute(steelBlue,0,ChartConstants.LS_SOLID, Color.white);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(pTransform1,
panelmeterattrib);
.
.
.
barplot.addPanelMeter(panelmeter);


RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(pTransform1,
paneltagmeterattrib, ChartConstants.RT_TAG_STRING);
panelmeter3.setPanelMeterPosition(ChartConstants.OUTSIDE_PLOTAREA_MAX);
panelmeter3.setTextColor(Color.black);
panelmeter3.getStringTemplate().setTextFont(font12);
panelmeter3.setAlarmIndicatorColorMode(ChartConstants.RT_INDICATOR_COLOR_NO_ALARM_
CHANGE);
barplot.addPanelMeter(panelmeter3);
```

Time/Date Panel Meter

Class RTTimePanelMeter

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTTimePanelMeter

The **RTTimePanelMeter** class displays the time/date value of the time stamp of the associated **RTProcessVar** object. It contains a template based on the **QCChart2D TimeLabel** class that is used to specify the font and time/date format information associated with the panel meter.

RTTimePanelMeter constructors

```
public RTTimePanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   ChartAttribute attrib
);

public RTTimePanelMeter(
   PhysicalCoordinates transform,
```

```
   ChartAttribute attrib
);
```

Parameters

transform

The coordinate system for the new **RTTimePanelMeter** object.

datasource

The process variable associated with the panel meter.

attrib

The color attributes of the panel meter indicator.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setTimeTemplate**(value) and value := **getTimeTemplate**();

TimeTemplate	Set/Get the **TimeLabel** template for the panel meter time/date value. The text properties associated with the panel meter are set using this property. In addition, the time or calendar format of the time/date value is also set here.

A complete listing of **RTTimePanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for RTTimePanelMeter

The panel meter below, extracted from the Treadmill example, method **InitializeElapsedTimePanelMeter**, adds an **RTTimePanelMeter** as an independent panel meter at the bottom of the display. In this example the plot area of the coordinate system is set for the position of the **RTTimePanelMeter** using **pTransform1.setGraphBorderDiagonal(..)**. It is positioned inside the plot area using the INSIDE_INDICATOR position constant. A string panel meter places a title above the time panel meter.

```
ChartView chartVu  = this;

CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);

pTransform1.setGraphBorderDiagonal(0.3, .85, 0.55, 0.96) ;

ChartAttribute panelmeterattrib = new
ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
RTTimePanelMeter panelmeter = new RTTimePanelMeter(pTransform1,
timeOfDay,panelmeterattrib);
panelmeter.setPanelMeterPosition(ChartConstants.INSIDE_INDICATOR);
panelmeter.getTimeTemplate().setTextFont(font36Numeric);
panelmeter.getTimeTemplate().setTimeFormat(ChartConstants.TIMEDATEFORMAT_24HMS);
panelmeter.setAlarmIndicatorColorMode(ChartConstants.RT_INDICATOR_COLOR_NO_ALARM_C
HANGE);
chartVu.addChartObject(panelmeter);

ChartAttribute panelmetertagattrib = new ChartAttribute
(beige,0,ChartConstants.LS_SOLID, beige);
RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(pTransform1, timeOfDay,
panelmetertagattrib, ChartConstants.RT_TAG_STRING);
panelmeter3.getStringTemplate().setTextFont(font10);
panelmeter3.setPanelMeterPosition(ChartConstants.ABOVE_REFERENCED_TEXT);
panelmeter3.setPositionReference(panelmeter);
panelmeter3.setTextColor(Color.black);
chartVu.addChartObject(panelmeter3);
```

Class RTElapsedTimePanelMeter

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTElapsedTimePanelMeter

The **RTElapsedTimePanelMeter** class displays the elapsed time value of the time stamp of the associated **RTProcessVar** object, interpreting the numeric value of the time stamp in milliseconds. It contains a template based on the **QCChart2D ElapsedTimeLabel** class that is used to specify the font and time/date format information associated with the panel meter.

RTTimePanelMeter constructors

```
public RTElapsedTimePanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   ChartAttribute attrib
);

public RTElapsedTimePanelMeter(
   PhysicalCoordinates transform,
   ChartAttribute attrib
);
```

Parameters

transform

The coordinate system for the new **RTTimePanelMeter** object.

datasource

The process variable associated with the panel meter.

attrib

The color attributes of the panel meter indicator.

Selected Public Instance Properties

ElapsedTimeTemplate	Set/Get the **ElapsedTimeLabel** template for the panel meter time/date value. The text properties associated with the panel meter are set using this property. In addition, the time or calendar format of the time/date value is also set here.

A complete listing of **RTElapsedTimePanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for RTElpasedTimePanelMeter

The panel meter below, extracted from the Treadmill example, method **InitializeStopWatchTimePanelMeter**, adds an **RTElapsedTimePanelMeter** as an independent panel meter at the bottom of the display. In this example the plot area of the coordinate system is set for the position of the **RTElapsedTimePanelMeter** using **pTransform1.setGraphBorderDiagonal(..)**. It is positioned inside the plot area using the INSIDE_INDICATOR position constant. A string panel meter places a title above the time panel meter.


```
ChartView chartVu  = this;
CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);

pTransform1.setGraphBorderDiagonal(0.51, .85, 0.76, 0.96) ;
```

```
ChartAttribute panelmeterattrib = new
ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
RTElapsedTimePanelMeter panelmeter = new RTElapsedTimePanelMeter(pTransform1,
stopWatch,panelmeterattrib);
panelmeter.setPanelMeterPosition(ChartConstants.INSIDE_INDICATOR);
panelmeter.getTimeTemplate().setTextFont(font36Numeric);
panelmeter.getTimeTemplate().setTimeFormat(ChartConstants.TIMEDATEFORMAT_24HMS);
panelmeter.getTimeTemplate().setDecimalPos ( 0);
panelmeter.setAlarmIndicatorColorMode(ChartConstants.RT_INDICATOR_COLOR_NO_ALARM_C
HANGE);
chartVu.addChartObject(panelmeter);
```

Form Control Panel Meter

Class RTFormControlPanelMeter

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTPanelMeter
RTFormControlPanelMeter

The **RTFormControlPanelMeter** encapsulates an **RTFormControl** object (buttons and track bars primarily, though others will also work) in a panel meter format. This allows it to use the **RTPanelMeter** positioning constants to position the form controls with respect to indicators, plot areas, text, and other panel meters.

.

RTFormControlPanelMeter constructors

```
public RTFormControlPanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   Control formcontrol,
   ChartAttribute attrib
);

public RTFormControlPanelMeter(
   PhysicalCoordinates transform,
   Control formcontrol,
   ChartAttribute attrib
);
```

Parameters

transform

The coordinate system for the new **RTFormControlPanelMeter** object.

formcontrol

A reference to the **Form Control** assigned to this panel meter.

datasource

The process variable associated with the control.

attrib

The color attributes of the panel meter indicator.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setControlSizeMode**(value) and value := **getControlSizeMode**();

ControlSizeMode	Set/Get to the size mode for the Control. Use one of the Control size mode constants: RT_ORIG_CONTROL_SIZE, RT_MIN_CONTROL_SIZE, RT_INDICATORRECT_CONTROL_SIZE.

A complete listing of **RTFormControlPanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for RTControlTrackbar in an RTFormControlPanelMeter

The panel meter below, extracted from the Treadmill example, method **InitializeLeftPanelMeters**, adds an **RTFormControlPanelMeter** as an independent panel meter at the left of the display. In this example the plot area of the coordinate system is set for the position of the **RTFormPanelMeter using pTransform1.setGraphBorderDiagonal(..)**. It is positioned inside the plot area using the CUSTOM_POSITION position constant. The lower left cornet of the form control is placed at the (0.0, 0.0) position of the plot area in PHYS_POS coordinates. The size of the form control is set to the size of the plot area, (width = 1.0, height = 1.0) in PHYS_POS coordinates.

```
Font trackbarfont = font64Numeric;
Font trackbarTitlefont = font12Bold;

ChartView chartVu  = this;

CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);

pTransform1.setGraphBorderDiagonal(0.01, .12, 0.07, 0.3) ;

ChartAttribute attrib1 = new ChartAttribute (lightBlue, 3,ChartConstants.LS_SOLID,
lightBlue);

runnersPaceTrackbar = new RTControlTrackBar(0.0, 15.0, 5.0, 1.0, 1);
runnersPaceTrackbar.setOrientation(RTControlTrackBar.VERTICAL);
runnersPaceTrackbar.setPaintTicks(true);
runnersPaceTrackbar.setPaintLabels(true);

runnersPaceTrackbar.setRTValue(3); // MUST USE RTValue to set double value
```

```
RTFormControlPanelMeter formControlTrackBar1 = new
RTFormControlPanelMeter(pTransform1, runnersPaceTrackbar, attrib1);
formControlTrackBar1.setRTDataSource(runnersPace);
formControlTrackBar1.setPanelMeterPosition(ChartConstants.CUSTOM_POSITION);
formControlTrackBar1.setLocation(0,0.0, ChartConstants.PHYS_POS);
formControlTrackBar1.setFormControlSize(new ChartDimension(1.0,1.0)); // Must be
in same units as SetLocation

ChartAttribute panelmeterattrib = new
ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
RTNumericPanelMeter panelmeter1 = new RTNumericPanelMeter(pTransform1,
runnersPace,panelmeterattrib);
panelmeter1.getNumericTemplate().setTextFont(trackbarfont);
panelmeter1.getNumericTemplate().setDecimalPos(1);
panelmeter1.setPanelMeterPosition(ChartConstants.RIGHT_REFERENCED_TEXT);
panelmeter1.setPositionReference( formControlTrackBar1);
formControlTrackBar1.addPanelMeter(panelmeter1);

ChartAttribute panelmetertagattrib = new
ChartAttribute(beige,0,ChartConstants.LS_SOLID, beige);
RTStringPanelMeter panelmeter2 = new RTStringPanelMeter(pTransform1, runnersPace,
panelmetertagattrib, ChartConstants.RT_TAG_STRING);
panelmeter2.setPositionReference(panelmeter1);
panelmeter2.getStringTemplate().setTextFont(trackbarTitlefont);
panelmeter2.setPanelMeterPosition(ChartConstants.BELOW_REFERENCED_TEXT);
panelmeter2.setTextColor(Color.black);
formControlTrackBar1.addPanelMeter(panelmeter2);

chartVu.addChartObject(formControlTrackBar1);

CartesianCoordinates pTransform2 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
pTransform2.setGraphBorderDiagonal(0.01, .41, 0.07, 0.59) ;

treadmillElevationTrackbar = new  RTControlTrackBar(0.0, 15.0, 5.0, 1.0, 1);
treadmillElevationTrackbar.setOrientation(RTControlTrackBar.VERTICAL);
treadmillElevationTrackbar.setRTValue(0); // MUST USE RTValue to set double value
treadmillElevationTrackbar.setPaintTicks(true);
treadmillElevationTrackbar.setPaintLabels(true);

RTFormControlPanelMeter formControlTrackBar2 = new
RTFormControlPanelMeter(pTransform2, treadmillElevationTrackbar, attrib1);
formControlTrackBar2.setRTDataSource(treadmillElevation);
formControlTrackBar2.setLocation(0,0.0);
formControlTrackBar2.setFormControlSize( new ChartDimension(1.0,1.0));

RTNumericPanelMeter panelmeter3 = new RTNumericPanelMeter(pTransform2,
runnersPace,panelmeterattrib);
panelmeter3.getNumericTemplate().setTextFont(trackbarfont);
panelmeter3.getNumericTemplate().setDecimalPos(1);
panelmeter3.setPanelMeterPosition(ChartConstants.RIGHT_REFERENCED_TEXT);
panelmeter3.setPositionReference( formControlTrackBar2);
formControlTrackBar2.addPanelMeter(panelmeter3);

RTStringPanelMeter panelmeter4 = new RTStringPanelMeter(pTransform2, runnersPace,
panelmetertagattrib, ChartConstants.RT_TAG_STRING);
panelmeter4.setPositionReference(panelmeter3);
panelmeter4.getStringTemplate().setTextFont(trackbarTitlefont);
panelmeter4.setPanelMeterPosition(ChartConstants.BELOW_REFERENCED_TEXT);
panelmeter4.setTextColor(Color.black);
formControlTrackBar2.addPanelMeter(panelmeter4);

chartVu.addChartObject(formControlTrackBar2);
```

6. Single Channel Bar Indicator

RTBarIndicator

An **RTBarIndicator** is used to display the current value of an **RTProcessVar** using the height or width of a bar. One end of each bar is always fixed at the specified base value. Bars can change their size either in vertical or horizontal direction. Sub types within the **RTBarIndicator** class support segmented bars, custom segmented bars with variable width segments, and pointer bar indicators. Panel meters can be attached to the bar indicator, where they provide text for numeric read outs, alarm warnings, descriptions and titles.

Bar Indicator

Class RTBarIndicator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTBarIndicator

The bar indicator is a relatively simple plot object that resides in the plot area of the specified coordinate system. It is usually combined with axes and axis labels, though this is not required. Since the bar indicator does not include axes or axis labels as option, it is up to the user to explicitly create axis and axis label objects for the bar indicator graph. The **QCChart2D** axis and axis labels routines make this easy to do.

RTBarIndicator constructors

```
public RTBarIndicator(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   double barwidth,
   double barbase,
   ChartAttribute attrib,
   int barjust,
   int barorient
);
```

Parameters

transform

The coordinate system for the new **RTBarIndicator** object.

datasource
The process variable associated with the bar indicator.
barwidth
The width of the bar in physical units .
barbase
The base of the bar in physical units.
attrib
The color attributes of the bar indicator.
barjust
The justification of bars. Use one of the bar justification constants: JUSTIFY_MIN, JUSTIFY_CENTER, JUSTIFY_MAX..
barorient
The orientation of the bar indicator: HORIZ_DIR or VERT_DIR.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAlarmIndicatorColorMode**(value) and value := **getAlarmIndicatorColorMode**();

AlarmIndicatorColorMode (inherited from **RTSingleValueIndicator**)	Get/Set whether the color of the indicator objects changes on an alarm. Use one of the constants: RT_INDICATOR_COLOR_NO_ALARM_CHANGE, RT_INDICATOR_COLOR_CHANGE_ON_ALARM.
BarDatapointLabelPosition (inherited from **ChartPlot**)	Bar plots that support the display of data point values have the option of displaying the data point's numeric values above the bar, below the bar, or centered in the bar. Use one of the data point label position constants: INSIDE_BAR, OUTSIDE_BAR, or CENTERED_BAR.
BarJust (inherited from **ChartPlot**)	Set/Get the justification of bars in bar graph plot objects. Use one of the bar justification constants: JUSTIFY_MIN, JUSTIFY_CENTER, or JUSTIFY_MAX.
BarOffset	Set/Get the bar offset from its fixed x or y value in physical units.
BarOrient (inherited from **ChartPlot**)	Set/Get the orientation (HORIZ_DIR or VERT_DIR) for bar plots.
BarSpacing (inherited from **RTPlot**)	Set/Get the spacing between adjacent items in multi-channel plots.
BarWidth (inherited from **ChartPlot**)	Set/Get the width of bars, in physical coordinates, for bar plots.
ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a **ChartAttribute** object.

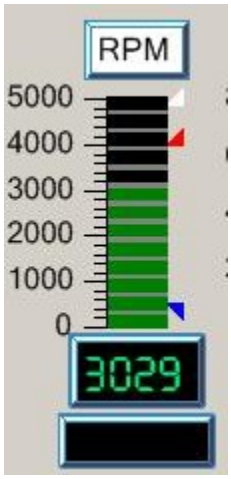
ChartObjClipping (inherited from **GraphObj**)	Sets the object clipping mode. Use one of the object clipping constants: NO_CLIPPING, GRAPH_AREA_CLIPPING, PLOT_AREA_CLIPPING, or INHERIT_CLIPPING.
ChartObjComponent (inherited from **GraphObj**)	Sets the reference to the **ChartView** component that the chart object is placed in
ChartObjEnable (inherited from **GraphObj**)	Enables/Disables the chart object. A chart object is drawn only if it is enabled. A chart object is enabled by default.
ChartObjScale (inherited from **GraphObj**)	Sets the reference to the PhysicalCoordinates object that the chart object is placed in
CurrentProcessValue (inherited from **RTSingleValueIndicator**)	Get the current process value of the primary channel.
FillBaseValue (inherited from **ChartPlot**)	Set/Get the base value, in physical coordinates, of solid (bars and filled areas) plot objects.
FillColor (inherited from **GraphObj**)	Sets the fill color for the chart object.
IndicatorBackground	Get/Set the background attribute of the bar indicator.
IndicatorBackgroundEnable	Set to true to enable the display of the bar indicator background.
IndicatorSubType	Get/Set the bar indicator sub type: RT_BAR_SOLID_SUBTYPE, RT_BAR_SEGMENTED_SUBTYPE, RT_BAR_SINGLE_SEGMENT_SUBTYPE, RT_POINTER_SUBTYPE.
LabelTemplateDecimalPos (inherited from **ChartPlot**)	Set/Get number of digits to the right of the decimal point in the **PlotLabelTemplate** property.
LabelTemplateNumericFormat (inherited from **ChartPlot**)	Set/Get the numeric format of the **PlotLabelTemplate** property.
LineColor (inherited from **GraphObj**)	Sets the line color for the chart object.
LineStyle (inherited from **GraphObj**)	Sets the line style for the chart object.
LineWidth (inherited from **GraphObj**)	Sets the line width for the chart object.
NumChannels (inherited from **RTPlot**)	Get the number of channels in the indicator.
PlotLabelTemplate (inherited from **ChartPlot**)	Set/Get the plot objects data point template. If the plot supports it, this **NumericLabel** object is used as a template to size, color and format the data point numeric values.
PointerSymbolNum	Set/Get the symbol used for the pointer symbol indicator subtype, RT_POINTER_SUBTYPE. Use one of the constants: RT_NO_SYMBOL, RT_LEFT_LOW_ALARM_SYMBOL.RT_LEFT_SET

	POINT_SYMBOL. RT_LEFT_HIGH_ALARM_SYMBOL , RT_RIGHT_LOW_ALARM_SYMBOL, RT_RIGHT_SETPOINT_SYMBOL, RT_RIGHT_HIGH_ALARM_SYMBOL, RT_TOP_LOW_ALARM_SYMBOL , RT_TOP_SETPOINT_SYMBOL, RT_TOP_HIGH_ALARM_SYMBOL, RT_BOTTOM_LOW_ALARM_SYMBOL, RT_BOTTOM_SETPOINT_SYMBOL, RT_BOTTOM_HIGH_ALARM_SYMBOL.
PrimaryChannel (inherited from **RTPlot**)	Set/Get the primary channel of the indicator.
RTDataSource (inherited from **RTSingleValueIndicator**)	Get/Set the array list holding the **RTProcessVar** variables for the indicator.
SegmentCornerRadius	Get/Set the corner radius used to draw the segment rounded rectangles.
SegmentSpacing	Get/Set the segments spacing for the RT_BAR_SEGMENTED_SUBTYPE and RT_BAR_SINGLE_SEGMENT_SUBTYPE bar indicator sub types.
SegmentValueRoundMode	Set/Get the segment value round mode. Specifies that the current process value is rounded up in calculating how many segments to display in RT_BAR_SEGMENTED_SUBTYPE and RT_BAR_SINGLE_SEGMENT_SUBTYPE modes. Use one of the constants: RT_FLOOR_VALUE, RT_CEILING_VALUE.
SegmentWidth	Get/Set the thickness of segments for the RT_BAR_SEGMENTED_SUBTYPE and RT_BAR_SINGLE_SEGMENT_SUBTYPE bar indicator sub types.
ShowDatapointValue (inherited from **ChartPlot**)	If the plot supports it, this method will turn on/off the display of data values next to the associated data point.
StepMode (inherited from **ChartPlot**)	Set/Get the plot objects step mode. Use one of the line plot step constants: NO_STEP, STEP_START, STEP_END, or STEP_NO_RISE_LINE.
ZOrder (inherited from **GraphObj**)	Sets the z-order of the object in the chart. Every object has a z-order value. Each z-order value does NOT have to be unique. If z-order sorting is turned on in the ChartView object, objects are sorted by z-order before they are drawn.

A complete listing of **RTBarIndicator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for an RTBarIndicator Segmented Bar Indicator

The bar indicator example below, extracted from the Dynamometer example, method **InitializeEngine1RPMIndicator**, creates the segmented bar RPM indicator in the upper left corner of the graph. It demonstrates how the plot area is defined for the bar indicator, how to create axes and axis labels. An **RTAlarmIndicator** is also created to display the alarm limit symbols to the right of the bar indicator. The image below also includes an **RTStringPanelMeter** for the "RPM" tag, an **RTNumericPanelMeter** for the numeric readout below the bar indicator, and an **RTAlarmPanelMeter** below that. See the Dynamometer example program for the complete program listing that creates all of these objects.



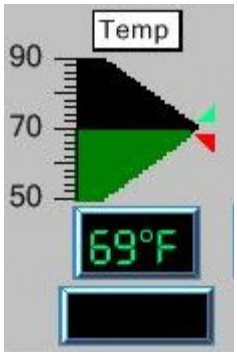

```
ChartView chartVu  = this;

CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 5000.0);

pTransform1.setGraphBorderDiagonal(0.06, 0.175, 0.08, 0.35) ;

Background background = new Background( pTransform1, ChartConstants.PLOT_BACKGROUND,
                              Color.gray);

chartVu.addChartObject(background);

ChartAttribute attrib1 = new ChartAttribute (Color.green, 1,ChartConstants.LS_SOLID,
                              Color.green);

double barwidth = 1.0, barbase = 0.0;

int barjust = ChartConstants.JUSTIFY_MIN;

int barorient = ChartConstants.VERT_DIR;

LinearAxis baraxis = new LinearAxis(pTransform1, ChartConstants.Y_AXIS);

chartVu.addChartObject(baraxis);
```

```
NumericAxisLabels barAxisLab = new NumericAxisLabels(baraxis);

chartVu.addChartObject(barAxisLab);


RTBarIndicator barplot = new RTBarIndicator(pTransform1,

        EngineRPM1,  barwidth,  barbase, attrib1,   barjust,  barorient);

        barplot.setSegmentSpacing(400);

barplot.setSegmentWidth( 250);

barplot.setIndicatorBackground( new ChartAttribute(Color.black, 1,
                              ChartConstants.LS_SOLID, Color.black));

barplot.setSegmentValueRoundMode( ChartConstants.RT_CEILING_VALUE);

barplot.setSegmentCornerRadius( 0);

barplot.setIndicatorSubType( ChartConstants.RT_BAR_SEGMENTED_SUBTYPE);

RTAlarmIndicator baralarms = new     RTAlarmIndicator(baraxis, barplot);

chartVu.addChartObject(baralarms);


ChartAttribute panelmeterattrib =

        new ChartAttribute(Color.blue,3,ChartConstants.LS_SOLID, Color.black);

ChartAttribute paneltagmeterattrib = new

        ChartAttribute(Color.blue,3,ChartConstants.LS_SOLID, Color.white);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(pTransform1, panelmeterattrib);

.

.

.

barplot.addPanelMeter(panelmeter);

RTAlarmPanelMeter panelmeter2 = new RTAlarmPanelMeter(pTransform1, panelmeterattrib);

.

.

.

barplot.addPanelMeter(panelmeter2);

RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(pTransform1, paneltagmeterattrib,
                              ChartConstants.RT_TAG_STRING);

.
```

```
.

.

barplot.addPanelMeter(panelmeter3);

chartVu.addChartObject(barplot);
```

Example for an RTBarIndicator Custom Segmented Bar Indicator

The custom bar indicator example below, extracted from the HomeAutomation example, method **ThermostatControl.InitializeCustomBarIndicator**, uses a segmented bar indicator to display the temperature. It uses a special feature that allows the width of the each bar segment to be calculated as a function of the height. This is done by subclassing the **RTBarIndicator** class and overriding the **getCustomBarWidth** and **getCustomBarOffset** methods. In the **CustomRTBarIndicator** example below, the width of the bar is calculated using a function based on the deviation of the current temperature from the *temperatureSetpoint* value. Calculating the bar width as a function of the bar height only works with the segmented bar subtypes. If you want a solid bar, make the **RTBarIndicator.setSegmentWidth** and **RTBarIndicator.setSegmetSpacing** values small, and the same, as in the example below.

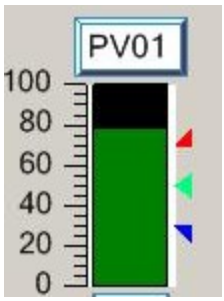


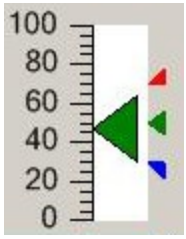

```
public class CustomRTBarIndicator extends  RTBarIndicator
{
    double temperatureSetpoint = 70;
    public CustomRTBarIndicator(PhysicalCoordinates transform,
        RTProcessVar datasource, double barwidth, double barbase,
    ChartAttribute attrib,  int barjust, int barorient)
    {
        super(transform, datasource, barwidth, barbase, attrib, barjust,
barorient);

    }

    public  double getCustomBarOffset(double v)
    {
        double offset = 0.0;
```

```
        return offset;
    }

    public  double getCustomBarWidth(double v)
    {
// Calculate width as fraction of initial bar width
    double width = 1.0;
// Bar widest at setpoint, narrowest at endpoints
// Clamp width to 0.05 to 1.0 range
    width = Math.max(0.05,this.getBarWidth() - Math.abs(0.04 * (v -
temperatureSetpoint)));
    width = Math.min(1.0,width);
    return width;
}

///<summary>
/// Set/Get local setpoint
///</summary>
    public double getTemperatureSetpoint()
    {
        return temperatureSetpoint;
    }
///<summary>
/// Set/Get local setpoint
///</summary>
    public void setTemperatureSetpoint(double value)
    {
         temperatureSetpoint = value;
    }
}
.


.


.

    barplot = new CustomRTBarIndicator(pTransform1,
        currentTemperature1,  barwidth,  barbase,
        attrib1,   barjust,  barorient);
    barplot.setIndicatorBackground(new ChartAttribute(Color.black, 1,
ChartConstants.LS_SOLID, Color.black));
    barplot.setSegmentSpacing(1);
    barplot.setSegmentWidth(1);
    barplot.setIndicatorSubType(ChartConstants.RT_BAR_SEGMENTED_SUBTYPE);
    barplot.setSegmentValueRoundMode(ChartConstants.RT_CEILING_VALUE);
.
.
.
    chartVu.addChartObject(barplot);
```

Example for an RTBarIndicator Solid Bar Indicator and Pointer Indicator

Setting up the solid bar and pointer indicators are pretty much identical to the segmented bar indicator. The examples below are extracted from the DynBarDemo example program, file DynSolidBars and DynPointers, method **InitializeBar1**. The default value for the **IndicatorSubType** property is RT_BAR_SOLID_SUBTYPE so that does not even need to be set.






```
// For solid bar indicator
ChartAttribute attrib1 =
       new ChartAttribute (Color.Black, 1,ChartConstants.LS_SOLID, Color.green);
double barwidth = 1.0, barbase = 0.0;
int barjust = ChartConstants.JUSTIFY_MIN;
int barorient = ChartConstants.VERT_DIR;
.
.
RTBarIndicator barplot = new RTBarIndicator(pTransform1,
       processVar1,  barwidth,  barbase,
       attrib1,   barjust,  barorient);
barplot.IndicatorSubType = ChartConstants.RT_BAR_SOLID_SUBTYPE;
.
.
chartVu.addChartObject(barplot);


// For Pointer indicator
ChartAttribute attrib1 =
       new ChartAttribute (Color.Black, 1,ChartConstants.LS_SOLID, Color.green);
double barwidth = 1.0, barbase = 0.0;
int barjust = ChartConstants.JUSTIFY_MIN;
int barorient = ChartConstants.VERT_DIR;
.
.
RTBarIndicator barplot = new RTBarIndicator(pTransform1,
       processVar1,  barwidth,  barbase,
       attrib1,   barjust,  barorient);
barplot.IndicatorSubType = ChartConstants.RT_POINTER_SUBTYPE;
.
.
chartVu.addChartObject(barplot);
       attrib1,   barjust,  barorient);
```

7. Multiple Channel Bar Indicator

RTMultiBarIndicator

An RTMultiBarIndicator is used to display the current value of a collection of RTProcessVar objects using a group of bars changing size. The bars are always fixed at the specified base value. Bars can change their size either in vertical or horizontal direction. Sub types within the RTMultiBarIndicator class support segmented bars, custom segmented bars with variable width segments, and pointer bar indicators.

Multiple Channel Bar Indicator

Class RTMultiBarIndicator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTMultiValueIndicator
RTMultiBarIndicator

The multi-bar indicator displays a collection of **RTProcessVar** objects that are related, or at the very least comparable, when bar graphed against one another using the same physical coordinate system.. It is usually combined with axes and axis labels, though this is not required. Since the bar indicator does not include axes or axis labels as option, it is up to the user to explicitly create axis and axis label objects for the bar indicator graph. The **QCChart2D** axis and axis labels routines make this easy to do.

When an **RTPanelMeter** object is added to an **RTMultiBarIndicator**, it is used as a template to create a multiple panel meters, one for each bar of the multi-bar indicator. The panel meter for each bar will reference the process variable information associated with that bar, stored in the **RTProcessVar** objects attached to the multi-bar indicator.

```
public RTMultiBarIndicator(
   PhysicalCoordinates transform,
   RTProcessVar[] datasource,
   double barwidth,
   double barspacing,
   double barbase,
   ChartAttribute[] attribs,
   int barjust,
   int barorient
);
```

Parameters

transform
The coordinate system for the new **RTMultiBarIndicator** object.
datasource
An array of the process variables associated with the bar indicator.
barwidth
The width of each bar in physical units.
barspacing
The space between adjacent bars in physical units.
barbase
The base of the bar in physical units.
attribs
An array of the color attributes of the bar indicator.
barjust
The justification of bars. Use one of the bar justification constants: JUSTIFY_MIN, JUSTIFY_CENTER, JUSTIFY_MAX..
barorient
The orientation of the bar indicator: HORIZ_DIR or VERT_DIR.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAlarmIndicatorColorMode**(value) and value := **getAlarmIndicatorColorMode**();

AlarmIndicatorColorMode (inherited from **RTSingleValueIndicator**)	Get/Set whether the color of the indicator objects changes on an alarm. Use one of the constants: RT_INDICATOR_COLOR_NO_ALARM_CHANGE, RT_INDICATOR_COLOR_CHANGE_ON_ALARM..
BarDatapointLabelPosition (inherited from **ChartPlot**)	Bar plots that support the display of data point values have the option of displaying the data point's numeric values above the bar, below the bar, or centered in the bar. Use one of the data point label position constants: INSIDE_BAR, OUTSIDE_BAR, or CENTERED_BAR.
BarJust (inherited from **ChartPlot**)	Set/Get the justification of bars in bar graph plot objects. Use one of the bar justification constants: JUSTIFY_MIN, JUSTIFY_CENTER, or JUSTIFY_MAX.
BarOffset	Set/Get the bar offset from its fixed x or y value in physical units.
BarOrient (inherited from **ChartPlot**)	Set/Get the orientation (HORIZ_DIR or VERT_DIR) for bar plots.
BarSpacing (inherited from **RTPlot**)	Set/Get the spacing between adjacent items in multi-channel plots.
BarWidth (inherited from **ChartPlot**)	Set/Get the width of bars, in physical coordinates, for bar plots.
ChartObjAttributes (inherited	Sets the attributes for a chart object using a

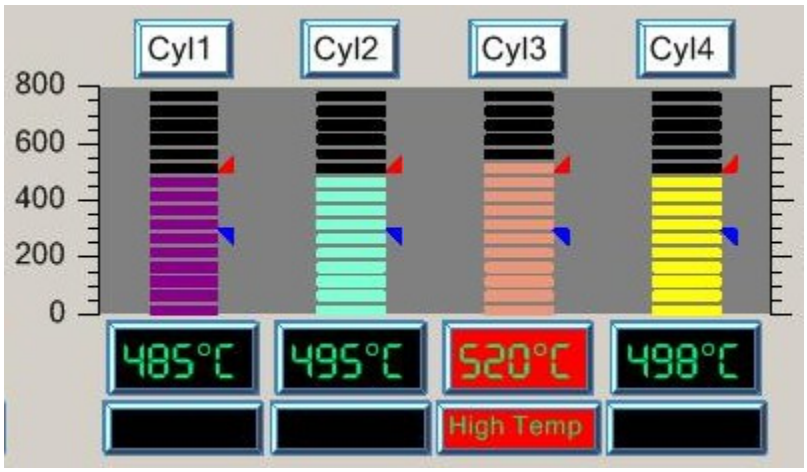
from **GraphObj**)	**ChartAttribute** object.
ChartObjClipping (inherited from **GraphObj**)	Sets the object clipping mode. Use one of the object clipping constants: NO_CLIPPING, GRAPH_AREA_CLIPPING, PLOT_AREA_CLIPPING, or INHERIT_CLIPPING.
ChartObjComponent (inherited from **GraphObj**)	Sets the reference to the **ChartView** component that the chart object is placed in
ChartObjEnable (inherited from **GraphObj**)	Enables/Disables the chart object. A chart object is drawn only if it is enabled. A chart object is enabled by default.
ChartObjScale (inherited from **GraphObj**)	Sets the reference to the **PhysicalCoordinates** object that the chart object is placed in
CurrentProcessValue (inherited from **RTSingleValueIndicator**)	Get the current process value of the primary channel.
FillBaseValue (inherited from **ChartPlot**)	Set/Get the base value, in physical coordinates, of solid (bars and filled areas) plot objects.
FillColor (inherited from **GraphObj**)	Sets the fill color for the chart object.
IndicatorBackground	Get/Set the background attribute of the bar indicator.
IndicatorBackgroundEnable	Set to true to enable the display of the bar indicator background.
IndicatorSubType	Get/Set the bar indicator sub type: RT_BAR_SOLID_SUBTYPE, RT_BAR_SEGMENTED_SUBTYPE, RT_BAR_SINGLE_SEGMENT_SUBTYPE, RT_POINTER_SUBTYPE.
LabelTemplateDecimalPos (inherited from **ChartPlot**)	Set/Get number of digits to the right of the decimal point in the **PlotLabelTemplate** property.
LabelTemplateNumericFormat (inherited from **ChartPlot**)	Set/Get the numeric format of the **PlotLabelTemplate** property.
LineColor (inherited from **GraphObj**)	Sets the line color for the chart object.
LineStyle (inherited from **GraphObj**)	Sets the line style for the chart object.
LineWidth (inherited from **GraphObj**)	Sets the line width for the chart object.
NumChannels (inherited from **RTPlot**)	Get the number of channels in the indicator.
PlotLabelTemplate (inherited from **ChartPlot**)	Set/Get the plot objects data point template. If the plot supports it, this **PlotLabelTemplate** object is used as a template to size, color and format the data point numeric values.
PointerSymbolNum	Set/Get the symbol used for the pointer symbol indicator subtype, RT_POINTER_SUBTYPE. Use one of the constants: RT_NO_SYMBOL,

	RT_LEFT_LOW_ALARM_SYMBOL.RT_LEFT_SET POINT_SYMBOL. RT_LEFT_HIGH_ALARM_SYMBOL , RT_RIGHT_LOW_ALARM_SYMBOL, RT_RIGHT_SETPOINT_SYMBOL, RT_RIGHT_HIGH_ALARM_SYMBOL, RT_TOP_LOW_ALARM_SYMBOL , RT_TOP_SETPOINT_SYMBOL, RT_TOP_HIGH_ALARM_SYMBOL, RT_BOTTOM_LOW_ALARM_SYMBOL, RT_BOTTOM_SETPOINT_SYMBOL, RT_BOTTOM_HIGH_ALARM_SYMBOL.
PrimaryChannel (inherited from **RTPlot**)	Set/Get the primary channel of the indicator.
RTDataSource (inherited from **RTSingleValueIndicator**)	Get/Set the array list holding the **RTProcessVar** variables for the indicator.
SegmentCornerRadius	Get/Set the corner radius used to draw the segment rounded rectangles.
SegmentSpacing	Get/Set the segments spacing for the RT_BAR_SEGMENTED_SUBTYPE and RT_BAR_SINGLE_SEGMENT_SUBTYPE bar indicator sub types.
SegmentValueRoundMode	Set/Get the segment value round mode. Specifies that the current process value is rounded up in calculating how many segments to display in RT_BAR_SEGMENTED_SUBTYPE and RT_BAR_SINGLE_SEGMENT_SUBTYPE modes. Use one of the constants: RT_FLOOR_VALUE, RT_CEILING_VALUE.
SegmentWidth	Get/Set the thickness of segments for the RT_BAR_SEGMENTED_SUBTYPE and RT_BAR_SINGLE_SEGMENT_SUBTYPE bar indicator sub types.
ShowDatapointValue (inherited from **ChartPlot**)	If the plot supports it, this method will turn on/off the display of data values next to the associated data point.
StepMode (inherited from **ChartPlot**)	Set/Get the plot objects step mode. Use one of the line plot step constants: NO_STEP, STEP_START, STEP_END, or STEP_NO_RISE_LINE.
ZOrder (inherited from **GraphObj**)	Sets the z-order of the object in the chart. Every object has a z-order value. Each z-order value does NOT have to be unique. If z-order sorting is turned on in the **ChartView** object, objects are sorted by z-order before they are drawn.

A complete listing of **RTMultiBarIndicator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for an RTMultiBarIndicator Segmented Bar Indicator

The multi-bar indicator example below, extracted from the Dynamometer example, method **InitializeEngine1TempIndicator**, creates the 4-bar segmented bar temperature indicator in the left upper section of the graph. It demonstrates how the plot area is defined for the multi-bar indicator, how to create axes and axis labels. An **RTAlarmIndicator** is also created to display the alarm limit symbols to the right of the bar indicator. The image below also includes an **RTStringPanelMeter** for the "Cyl#" tag, an **RTNumericPanelMeter** for the numeric readout below each bar indicator, and an **RTAlarmPanelMeter** below that. See the Dynamometer example program for the complete program listing that creates all of these objects.



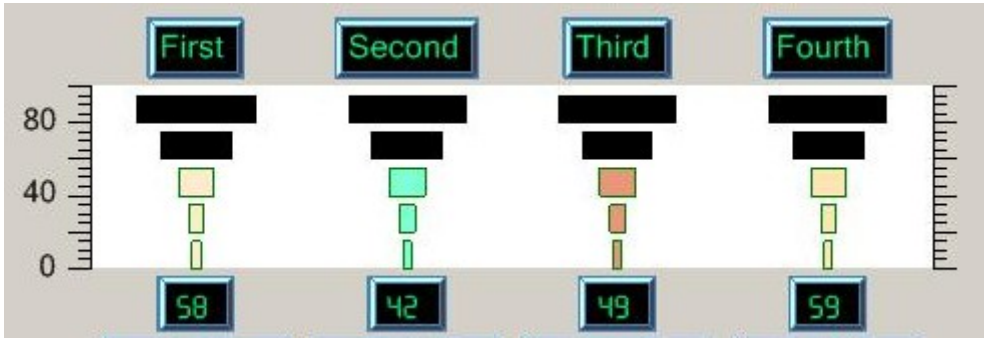

```
private void InitializeEngine1TempIndicator()
{

    ChartView chartVu  = this;
    CartesianCoordinates pTransform1 =
        new CartesianCoordinates( 0.0, 0.0, 1.0, 800.0);

    pTransform1.setGraphBorderDiagonal(0.15, .175, 0.48, 0.35) ;

    Background background = new Background( pTransform1,
        ChartConstants.PLOT_BACKGROUND, Color.gray);
    chartVu.addChartObject(background);

    ChartAttribute attrib1 = new ChartAttribute (Color.red,
        1,ChartConstants.LS_SOLID, Color.red);
    ChartAttribute attrib2 = new ChartAttribute (Color.blue,
        1,ChartConstants.LS_SOLID, Color.blue);
    ChartAttribute attrib3 = new ChartAttribute (Color.green,
        1,ChartConstants.LS_SOLID, Color.green);
    ChartAttribute attrib4 = new ChartAttribute (Color.yellow,
        1,ChartConstants.LS_SOLID, Color.yellow);

    ChartAttribute []attribArray = {attrib1, attrib2, attrib3, attrib4};

    double barwidth = 0.1, barbase = 0.0, barspace = 0.25;
    int barjust = ChartConstants.JUSTIFY_MIN;
    int barorient = ChartConstants.VERT_DIR;

    LinearAxis baraxis = new LinearAxis(pTransform1, ChartConstants.Y_AXIS);
```

```
    chartVu.addChartObject(baraxis);

    LinearAxis baraxis2 = new LinearAxis(pTransform1, ChartConstants.Y_AXIS);
    baraxis2.setAxisIntercept(pTransform1.getStopX());
    baraxis2.setAxisTickDir(ChartConstants.AXIS_MAX);
    chartVu.addChartObject(baraxis2);

    NumericAxisLabels barAxisLab = new NumericAxisLabels(baraxis);
    chartVu.addChartObject(barAxisLab);

// This uses an RTMultiProcessVar (EngineCylinders1) to initialize the indicator
    RTMultiBarIndicator barplot = new RTMultiBarIndicator(pTransform1,
            EngineCylinders1,  barwidth, barspace, barbase,
            attribArray,  barjust,  barorient);

    barplot.setSegmentSpacing(50);
    barplot.setSegmentWidth( 30);
    barplot.setIndicatorBackground( new ChartAttribute(Color.black, 1,
        ChartConstants.LS_SOLID, Color.black));
    barplot.setSegmentValueRoundMode( ChartConstants.RT_CEILING_VALUE);
    barplot.setSegmentCornerRadius( 0);
    barplot.setIndicatorSubType( ChartConstants.RT_BAR_SEGMENTED_SUBTYPE);
.
.
.

    RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(pTransform1,
panelmeterattrib);
.
.
    barplot.addPanelMeter(panelmeter);

    RTAlarmPanelMeter panelmeter2 = new RTAlarmPanelMeter(pTransform1,
panelmeterattrib);
.
.

    barplot.addPanelMeter(panelmeter2);

    RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(pTransform1,
paneltagmeterattrib, ChartConstants.RT_TAG_STRING);
.
.
 ChartConstants.RT_INDICATOR_COLOR_NO_ALARM_CHANGE);
    barplot.addPanelMeter(panelmeter3);


    chartVu.addChartObject(barplot);
}
```

Example for an RTMultiBarIndicator Custom Segmented Bar Indicator

The custom bar indicator example below is extracted from the DynBarDemo example, file **DynCustomBars**, method **InitializeBar3**. It uses a special feature that allows the width of the each bar segment to be calculated as a function of the height. This is done by subclassing the **RTBarIndicator** and **RTMultiBarIndicator** classes and overriding the **getCustomBarWidth** and **getCustomBarOffset** methods. In the example below, the width and offset of the bar is calculated using a function based on the height value. Calculating the bar width as a function of the bar height only works with the segmented bar subtypes. If you want a solid bar with custom widths, make the

RTBarIndicator.setSegmentWidth and **RTBarIndicator.setSegmetSpacing** values equal and small.




```
public class CustomRTBarIndicator extends  RTBarIndicator
{
    public CustomRTBarIndicator(PhysicalCoordinates transform,
        RTProcessVar datasource, double barwidth, double barbase,
        ChartAttribute attrib,  int barjust, int barorient)
    {
    super(transform, datasource, barwidth, barbase, attrib, barjust, barorient);
    }

    public  double getCustomBarOffset(double v)
    {
        double offset = 0.0;
        return offset;
    }

    public  double getCustomBarWidth(double v)
    {
        // Calculate width as fraction of initial bar width
        double width = 0.5;
        width = 0.01 + (v/100) * (v/100) * this.getBarWidth();
        return width;
    }
}

private void InitializeBar3()
{
    RTProcessVar [] processVarArray =
        {processVar1, processVar2, processVar3, processVar4};

    ChartView chartVu  = this;

    CartesianCoordinates pTransform1 =
        new CartesianCoordinates( 0.0, 0.0, 1.0, 100.0);

    pTransform1.setGraphBorderDiagonal(0.05, .475, 0.5, 0.65) ;

    Background background = new Background( pTransform1,
        ChartConstants.PLOT_BACKGROUND, Color.white);
    chartVu.addChartObject(background);

    ChartAttribute attrib1 = new ChartAttribute (Color.red,
        1,ChartConstants.LS_SOLID, Color.red);
    ChartAttribute attrib2 = new ChartAttribute (Color.blue,
        1,ChartConstants.LS_SOLID, Color.blue);
    ChartAttribute attrib3 = new ChartAttribute (Color.green,
        1,ChartConstants.LS_SOLID, Color.green);
    ChartAttribute attrib4 = new ChartAttribute (Color.yellow,
        1,ChartConstants.LS_SOLID, Color.yellow);
```

```
ChartAttribute []attribArray = {attrib1, attrib2, attrib3, attrib4};

double barwidth = 0.20, barbase = 0.0, barspace = 0.25;
int barjust = ChartConstants.JUSTIFY_MIN;
int barorient = ChartConstants.VERT_DIR;

LinearAxis baraxis = new LinearAxis(pTransform1, ChartConstants.Y_AXIS);
chartVu.addChartObject(baraxis);

LinearAxis baraxis2 = new LinearAxis(pTransform1, ChartConstants.Y_AXIS);
baraxis2.setAxisIntercept(pTransform1.getStopX());
baraxis2.setAxisTickDir(ChartConstants.AXIS_MAX);
chartVu.addChartObject(baraxis2);

NumericAxisLabels barAxisLab = new NumericAxisLabels(baraxis);
chartVu.addChartObject(barAxisLab);

CustomRTMultiBarIndicator barplot = new CustomRTMultiBarIndicator(pTransform1,
    processVarArray,  barwidth, barspace, barbase,
    attribArray,  barjust,  barorient);
barplot.setSegmentSpacing(20);
barplot.setSegmentWidth(15);

ChartAttribute panelmeterattrib = new ChartAttribute(Color.lightGray,
    3,ChartConstants.LS_SOLID, Color.black);

RTNumericPanelMeter panelmeter =
    new RTNumericPanelMeter(pTransform1, processVar1,panelmeterattrib);
panelmeter.setPanelMeterPosition(ChartConstants.OUTSIDE_PLOTAREA_MIN);
panelmeter.getNumericTemplate().setTextFont(font12Numeric);
barplot.addPanelMeter(panelmeter);

RTAlarmPanelMeter panelmeter2 = new RTAlarmPanelMeter(pTransform1,
    processVar1,panelmeterattrib);
.
.
barplot.addPanelMeter(panelmeter2);

RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(pTransform1,
    processVar1, panelmeterattrib, ChartConstants.RT_TAG_STRING);
.
.
barplot.addPanelMeter(panelmeter3);

barplot.setIndicatorSubType(RT_BAR_SEGMENTED_SUBTYPE);
barplot.setSegmentValueRoundMode(ChartConstants.RT_CEILING_VALUE);

chartVu.addChartObject(barplot);
}
```

8. Meters Coordinates, Meter Axes and Meter Axis Labels

RTMeterCoordinates
RTMeterAxis
RTMeterAxisLabels
RTMeterStringAxisLabels

Familiar examples of analog meter indicators are voltmeters, car speedometers, pressure gauges, compasses and analog clock faces. A meter usually consists of a meter coordinate system, meter axes, meter axis labels, and a meter indicator (the needle, arc or symbol used to display the current value). It can also have panel meters (**RTPanelMeter** derived objects) that display the meter title, numeric readout and alarm state. The first three objects, the meter coordinate system, meter axis and meter axis labels are described in this chapter, while the meter indicator types are described in the next.

Meter Coordinates

Class RTMeterCoordinates

QChart2D.PolarCoordinates
RTMeterCoordinates

A meter coordinate system has more properties than a simple Cartesian coordinate system, or even a polar coordinate system. Because of the variation in meter styles, a meter coordinate system needs to define the start and end angle of the meter arc within the 360 degree polar coordinate system. It also needs to map a physical coordinate system, representing the meter scale, on top of the meter arc. And the origin of the meter coordinate system can be offset in both x- and y-directions with respect to the containing plot area.

RTMeterCoordinates constructors

```
public RTMeterCoordinates(
   double startarcangle,
   double arcextent,
   double startarcscale,
   double endarcscale,
   boolean arcdirection,
   double x,
   double y,
   double arcradius
);
```

```
public RTMeterCoordinates(
   double startarcangle,
   double arcextent,
   double startarcscale,
   double endarcscale,
   boolean arcdirection,
   double arcradius
);
```

Parameters

startarcangle
: Specifies the starting arc angle position of the meter arc in degrees.

arcextent
: Specifies the extent of the meter arc in degrees. The default meter arc starts at *startArcAngle* and extends in a negative (clockwise) direction with an extent *arcExtent*.

startarcscale
: Specifies the scaling value associated with the *startArcAngle* position of the meter arc.

endarcscale
: Specifies the scaling value associated with the ending position of the meter arc.

arcdirection
: Specifies the direction of the *arcextent*. The default *arcDirectionPositive* value of false meter arc starts at *startArcAngle* and extends in a negative (clockwise) direction with an extent *arcExtent*. Change to true to have the meter arc extend in a positive (counter-clockwise) direction.

x
: Specifies x-position of the center of the meter arc in plot area normalized coordinates.

y
: Specifies y-position of the center of the meter arc in plot area normalized coordinates.

arcradius
: Specifies radius of the meter arc in plot area normalized coordinates.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setArcCenterX**(value) and value := **getArcCenterX**();

ArcCenterX	Get/Set Specifies x-position of the center of the meter arc in plot area normalized coordinates.

ArcCenterY	Get/Set Specifies y-position of the center of the meter arc in plot area normalized coordinates.
ArcDirectionPositive	Get/Set the direction of the *arcExtent*. The default *arcDirectionPositive* value of false meter arc starts at *startArcAngle* and extends in a negative (clockwise) direction with an extent *arcExtent*. Change to true to have the meter arc extend in a positive (counter-clockwise) direction.
ArcExtent	Specifies the extent of the meter arc in degrees. The default meter arc starts at *startArcAngle* and extends in a negative (clockwise) direction with an extent *arcExtent*.
ArcRadius	Get/Set radius of the meter arc in plot area normalized coordinates.
EndArcScale	Get/Set the scaling value associated with the ending position of the meter arc.
StartArcAngle	Get/Set Specifies the starting arc angle position of the meter arc in degrees.
StartArcScale	Get/Set the scaling value associated with the *startArcAngle* position of the meter arc.

A complete listing of **RTMeterCoordinates** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Examples for meter coordinate system

The example below, extracted from the WeatherStation example, method **InitializeHumidity**, creates a meter coordinate system that starts at the arc angle of 225 degrees and has an arc extent of 270 degrees. The meter scale extends from 0.0 to 100.0 in the counterclockwise direction.


```
// Fahrenheight scale
double startarcangle = 225;
double arcextent = 270;
double startarcscale = 0.0;
double endarcscale = 100.0;
boolean arcdirection = false;
double arcradius = 0.6;
double centerx = 0.0, centery= 0.2;
Font meterFont = font12;


RTMeterCoordinates meterframe1 = new RTMeterCoordinates(startarcangle,
    arcextent, startarcscale, endarcscale, arcdirection, centerx,
    centery, arcradius);
```

The example below, extracted from the MeterDemo example, file ArrowMeter, method **InitializeMeter4**, creates a meter coordinate system that starts at the arc angle of 90 degrees and has an arc extent of 180 degrees. The meter scale extends from 0.0 to 100.0 in the counterclockwise direction.

```
double startarcangle = 90;
double arcextent = 180;
double startarcscale = 0.0;
double endarcscale = 100.0;
boolean arcdirection = false;
double arcradius = 0.6;
double centerx = 0.25, centery= 0.0;
Font meterFont =font12;

RTMeterCoordinates meterframe = new RTMeterCoordinates(startarcangle, arcextent,
        startarcscale, endarcscale, arcdirection, centerx, centery, arcradius);
```

Meter Axis

RTMeterAxis

Com.quinncurtis.chart2djava.LinearAxis
RTMeterAxis

A meter axis extends for the extent of the meter arc and is centered on the origin. Major and minor tick marks are placed at evenly spaced intervals perpendicular to the meter arc. The meter axis also draws meter alarm arcs using the alarm information in the associated **RTProcessVar** object.

RTMeterAxis Constructors

```
public RTMeterAxis(
   RTMeterCoordinates frame,
   RTMeterIndicator graphplot
);

public RTMeterAxis(
   RTMeterCoordinates frame,
   RTMeterIndicator graphplot
   double tickspace,
   int tickspermajor

);
```

Parameters

frame

The **RTMeterCoordinates** object defining the meter properties for the meter axis.

graphplot

The **RTMeterIndicator** object associated with the meter axis.

tickspace

Specifies the spacing between minor tick marks, in degrees.

tickspermajor

Specifies the number of minor tick marks per major tick mark.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAxisLabels**(value) and value := **getAxisLabels**();

AxisLabels (inherited from **Axis**)	Get/Set the axis labels object associated with this axis.
AxisLineEnable (inherited from **Axis**)	Set/Get to true draws the axis line connecting the tick marks.
AxisMajorTickLength (inherited from **Axis**)	Get/Set length of a major tick mark.
AxisMax (inherited from **Axis**)	Get/Set the axis maximum value.
AxisMin (inherited from **Axis**)	Get/Set the axis minimum value.
AxisMinorTickLength (inherited from **Axis**)	Get/Set length of a minor tick mark.
AxisMinorTicksPerMajor (inherited from **Axis**)	Get/Set the number of minor tick marks per major tick mark.
AxisTickDir (inherited from **Axis**)	Get/Set the direction of a tick mark. Use one of the tick direction constants: AXIS_MIN, AXIS_CENTER, AXIS_MAX.
AxisTickOrigin (inherited from **Axis**)	Get/Set the starting point for positioning tick marks, in physical coordinates.
AxisTicksEnable (inherited from **Axis**)	Set/Get to true draws the axis tick marks.
AxisTickSpace (inherited from **LinearAxis**)	Get/Set the minor tick mark spacing.
ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a **ChartAttribute** object.
InnerAlarmArcNormalized	Get/Set the inner arc of the axis in normalized radius coordinates.
MajorTickLineWidth	Get/Set the major tick line width.
MeterAxisLabels	Get/Set the **RTMeterAxisLabels** object, if any, associated with this object.
MeterFrame	Get/Set the **RTMeterCoordinates** coordinate system associated with this object.
MeterIndicator	Get/Set the **RTMeterIndicator** object, if any, associated with this object.
MinorTickLineWidth	Get/Set the minor tick line width.
OuterAlarmArcNormalized	Get/Set the outer arc of the axis in

	normalized radius coordinates.
ShowAlarms	Get/Set true to show the alarm arcs.
ZOrder (inherited from **GraphObj**)	Sets the z-order of the object in the chart. Every object has a z-order value. Each z-order value does NOT have to be unique. If z-order sorting is turned on in the **ChartView** object, objects are sorted by z-order before they are drawn.

A complete listing of **RTMeterAxis** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for meter axis

The example below, extracted from the AutoInstrumentPanel example, method **InitializeTach**, creates a meter coordinate system that starts at the arc angle of 135 degrees and has an arc extent of 230 degrees. The meter scale extends from 0.0 to 8.0 in the counterclockwise direction. Two axes are created. The first is created so that it draws just the major tick marks using a thicker line width. The second uses thin tick marks for the minor tick marks of the meter axis.


```
ChartView chartVu  = this;
ChartAttribute attrib1 = new ChartAttribute (Color.black,
    1,ChartConstants.LS_SOLID, Color.blue);
double startarcangle = 135;
double arcextent = 230;
double startarcscale = 0.0;
double endarcscale = 8.0;
boolean arcdirection = false;
```

```
double arcradius = 0.75;
double centerx = 0.0, centery= -0.0;

RTMeterCoordinates meterframe = new RTMeterCoordinates(startarcangle, arcextent,
        startarcscale, endarcscale, arcdirection, centerx, centery, arcradius);

meterframe.setGraphBorderDiagonal(0.45, 0.2, 0.75, 0.9) ;

RTMeterNeedleIndicator meterneedle = new RTMeterNeedleIndicator(meterframe, tach);
           meterneedle.setChartObjAttributes(attrib1);
           meterneedle.setNeedleLength( 0.75);

ChartAttribute panelmeterattrib =
    new ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);

RTNumericPanelMeter panelmeter =
    new RTNumericPanelMeter(meterframe, tach, panelmeterattrib);
panelmeter.setPanelMeterPosition( ChartConstants.RADIUS_CENTER);
panelmeter.getNumericTemplate().setXJust(ChartConstants.JUSTIFY_MAX);
panelmeter.getNumericTemplate().setYJust( ChartConstants.JUSTIFY_MAX);
panelmeter.getNumericTemplate().setDecimalPos( 1);
panelmeter.setPanelMeterNudge( -4, 4);
panelmeter.setTextColor( springGreen);
panelmeter.getNumericTemplate().setTextFont(font24Numeric);
meterneedle.addPanelMeter(panelmeter);

ChartAttribute panelmetertagattrib =
    new ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.white);
RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(meterframe,
    tach, panelmetertagattrib, ChartConstants.RT_UNITS_STRING);
panelmeter3.setPanelMeterPosition(  ChartConstants.RADIUS_CENTER);
           panelmeter3.setPanelMeterNudge(0,-8);
panelmeter3.getStringTemplate().setXJust( ChartConstants.JUSTIFY_CENTER);
panelmeter3.getStringTemplate().setYJust( ChartConstants.JUSTIFY_MIN);
panelmeter3.setTextColor( Color.white);
panelmeter3.setFrame3DEnable( false);
panelmeter3.getStringTemplate().setTextBgMode( false);
panelmeter3.getStringTemplate().setTextFont(font12Bold);
meterneedle.addPanelMeter(panelmeter3);

chartVu.addChartObject(meterneedle);

RTMeterAxis meteraxis = new RTMeterAxis(meterframe, meterneedle);
meteraxis.setChartObjAttributes(attrib1);

meteraxis.setAxisTickDir(ChartConstants.AXIS_MIN);
meteraxis.setLineWidth( 5);
meteraxis.setColor( Color.white);
meteraxis.setAxisTickSpace(1);
meteraxis.setAxisMinorTicksPerMajor(1);
meteraxis.setShowAlarms( true);
meterneedle.setMeterAxis( meteraxis);
chartVu.addChartObject(meteraxis);

Font meterFont = font14Bold;
RTMeterAxisLabels meteraxislabels = new RTMeterAxisLabels(meteraxis);
meteraxislabels.setTextFont(meterFont);
meteraxislabels.setColor( Color.white);
meteraxislabels.setAxisLabelsDir(meteraxis.getAxisTickDir());
meteraxislabels.setOverlapLabelMode( ChartConstants.OVERLAP_LABEL_DRAW);
           chartVu.addChartObject(meteraxislabels);

RTMeterAxis meteraxis2 = new RTMeterAxis(meterframe, meterneedle);
meteraxis2.setChartObjAttributes(attrib1);
meteraxis2.setAxisTickDir(ChartConstants.AXIS_MIN);
meteraxis2.setLineWidth( 1);
meteraxis2.setColor( Color.white);
meteraxis2.setAxisTickSpace(0.1);
meteraxis2.setAxisMinorTickLength( 10);
meteraxis2.setAxisMajorTickLength( 10);
meteraxis2.setAxisMinorTicksPerMajor(10);
```

```
meteraxis2.setShowAlarms( false);
chartVu.addChartObject(meteraxis2);
```

Numeric Meter Axis Labels

Class RTMeterAxisLabels

Com.quinncurtis.chart2djava.NumericAxisLabels
RTMeterAxisLabels

This class labels the major tick marks of the **RTMeterAxis** class. The class supports many predefined and user-definable formats, including numeric, exponent, percentage, business and currency formats.

RTMeterAxisLabels constructor

```
public RTMeterAxisLabels(
   RTMeterAxis baseaxis
);
```

Parameters

baseaxis

The **RTMeterAxis** object associated with the labels.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAxisLabelsDecimalPos**(value) and value := **getAxisLabelsDecimalPos**();

AxisLabelsDecimalPos (inherited from **NumericAxisLabels**)	Set/Get the number of digits to the right of the decimal point for numeric axis labels.
AxisLabelsDir (inherited from **AxisLabels**)	Set/Get the justification of the axis labels with respect to the axis tick marks. Use one of the tick direction constants: AXIS_MIN, AXIS_MAX.
AxisLabelsEnds (inherited from **AxisLabels**)	Set/Get whether there should be labels for the axis minimum (LABEL_MIN), maximum (LABEL_MAX) or tick mark starting point (LABEL_ORIGIN). The value of these

	constants can be OR'd together. The value of LABEL_MIN | LABEL_MAX | LABEL_ORIGIN is LABEL_ALL
AxisLabelsFormat (inherited from **AxisLabels**)	Set/Get the numeric format for the axis labels.
AxisLabelsTickOffsetX (inherited from **AxisLabels**)	Set/Get the x-offset, in window device coordinates, of the label offset from the endpoint of the associated tick mark.
AxisLabelsTickOffsetY (inherited from **AxisLabels**)	Set/Get the y-offset, in window device coordinates, of the label offset from the endpoint of the associated tick mark.
ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a **ChartAttribute** object.
MeterAxis	Get/Set the **RTMeterAxis** associated with this object.
OverlapLabelMode (inherited from **AxisLabels**)	It is possible that axis labels overlap if the window that the axes are placed in is to small, the major tick marks are to close together, or in the case of time axis labels, to large for the current tick mark spacing. A test can be performed in the software to not display labels to overlap.
TextBgColor (inherited from **ChartText**)	Set/Get the color of the background rectangle under the text, if the textBgMode is true.
TextBgMode (inherited from **ChartText**)	Set/Get the text background color mode.
TextBoxColor (inherited from **ChartText**)	Set/Get if the text bounding box is drawn in the text box color.
TextBoxMode (inherited from **ChartText**)	Set/Get the text bounding box color.
TextFont (inherited from **ChartText**)	Set/Get the font of the text.
TextNudge (inherited from **ChartText**)	Set/Get the xy values of the textNudge property. It moves the relative position, using window device coordinates, of the text relative to the specified location of the text.
TextRotation (inherited from **ChartText**)	Set/Get the rotation of the text in the normal viewing plane.
ZOrder (inherited from **GraphObj**)	Sets the z-order of the object in the chart. Every object has a z-order value. Each z-order value does NOT have to be unique. If z-order sorting is turned on in the **ChartView** object, objects are sorted by z-order before they are drawn.

A complete listing of **RTMeterAxisLabels** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for meter axis labels

The example below, extracted from the WeatherStation example, method **InitializeHumidity**, creates a meter coordinate system that starts at the arc angle of 225 degrees and has an arc extent of 270 degrees. The meter scale extends from 0.0 to 100.0 in the counterclockwise direction. Two axes are created. The first is created so that it draws just the major tick marks using a thicker line width. The second uses thin tick marks for the minor tick marks of the meter axis. Only the first is included below since it is the one labeled.


```
ChartView chartVu  = this;

ChartAttribute attrib1 =
    new ChartAttribute (Color.black, 1,ChartConstants.LS_SOLID, Color.blue);

// Fahrenheight scale
double startarcangle = 225;
double arcextent = 270;
double startarcscale = 0.0;
double endarcscale = 100.0;
boolean arcdirection = false;
double arcradius = 0.6;
double centerx = 0.0, centery= 0.2;
Font meterFont = font12;

RTMeterCoordinates meterframe1 = new RTMeterCoordinates(startarcangle, arcextent,
    startarcscale, endarcscale, arcdirection, centerx, centery, arcradius);

ChartRectangle2D normrect = new ChartRectangle2D(0.67, 0.05, 0.32, 0.53);
RT3DFrame frame3d = new RT3DFrame(meterframe1, normrect, facePlateAttrib,
    ChartConstants.NORM_GRAPH_POS);
chartVu.addChartObject(frame3d);

meterframe1.setGraphBorderDiagonal(0.67, 0.05, 0.99, 0.58) ;

RTMeterNeedleIndicator meterneedle =
    new RTMeterNeedleIndicator(meterframe1, humidity);
meterneedle.setChartObjAttributes(attrib1);

meterneedle.setNeedleLength(0.6);

.
.
.
```

```
chartVu.addChartObject(meterneedle);

RTMeterAxis meteraxis1 = new RTMeterAxis(meterframe1, meterneedle);
meteraxis1.setChartObjAttributes(attrib1);

meteraxis1.setAxisTickDir(ChartConstants.AXIS_MIN);
meteraxis1.setLineWidth(3);
meteraxis1.setLineColor(Color.black);
meteraxis1.setAxisTickSpace(20);
meteraxis1.setAxisMinorTicksPerMajor(1);
meteraxis1.setShowAlarms(false);
meterneedle.setMeterAxis(meteraxis1);
chartVu.addChartObject(meteraxis1);

RTMeterAxisLabels meteraxislabels1 = new RTMeterAxisLabels(meteraxis1);
meteraxislabels1.setTextFont(meterFont);
meteraxislabels1.setAxisLabelsDir(meteraxis1.getAxisTickDir());
meteraxislabels1.setOverlapLabelMode(ChartConstants.OVERLAP_LABEL_DRAW);
chartVu.addChartObject(meteraxislabels1);

RTMeterAxis meteraxis2 = new RTMeterAxis(meterframe1, meterneedle);
meteraxis2.setChartObjAttributes(attrib1);
meteraxis2.setAxisTickDir(ChartConstants.AXIS_MIN);
meteraxis2.setLineWidth(1);
meteraxis2.setLineColor(Color.black);
meteraxis2.setAxisTickSpace(5);
meteraxis2.setAxisMinorTicksPerMajor(4);
meteraxis2.setShowAlarms(false);
meteraxis2.setAxisMinorTickLength(10);
chartVu.addChartObject(meteraxis2);
```

String Meter Axis Labels

Class RTMeterStringAxisLabels

Com.quinncurtis.chart2djava.StringAxisLabels
RTMeterStringAxisLabels

This class labels the major tick marks of the **RTMeterAxis** class using user-defined strings

RTMeterStringAxisLabels constructor

```
public RTMeterStringAxisLabels(
   RTMeterAxis baseaxis
);
```

Parameters

baseaxis

The **RTMeterAxis** object associated with the labels.

Parameters

baseaxis

The **RTMeterAxis** object associated with the labels.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setMeterLabelTextOrient**(value) and value := **getMeterLabelTextOrient**();

The properties of the **RTMeterStringAxisLabels** class are pretty much the same as the **RTMeterAxisLabels** class, with these exceptions.

MeterLabelTextOrient	Get/Set if the text is horizontal (METER_LABEL_HORIZONTAL) at right angles to the tick mark (METER_LABEL_PERPENDICULAR), or radial to the tick mark parallel (METER_LABEL_RADIAL_1, METER_LABEL_RADIAL_2).

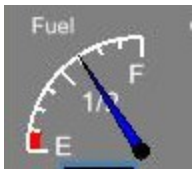
The axis label strings are set using the **setAxisLabelString** method.

setAxisLabelsStrings (inherited from **StringAxisLabels**)	Sets the string array used to hold user defined axis label strings. Setting the string array does not automatically turn on the use of string labels. Use **enableAxisLabelsStrings** to enable axis strings.

A complete listing of **RTMeterStringAxisLabels** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for string meter axis labels

The example below, extracted from the AutoInstrumentPanel example, method **InitializeFuel**, creates a meter coordinate system that starts at the arc angle of 180 degrees and has an arc extent of 90 degrees. The meter scale extends from 0.0 to 32.0 in the counterclockwise direction. The meter axis is labeled at the major tick marks with the strings {"E", "1/2", "F"}.




```
ChartView chartVu  = this;

ChartAttribute attrib1 = new ChartAttribute (Color.black,
        1,ChartConstants.LS_SOLID, Color.blue);
double startarcangle = 180;
double arcextent = 90;
double startarcscale = 0.0;
double endarcscale = 32.0;
boolean arcdirection = false;
double arcradius = 0.8;
double centerx = 0.0, centery= -0.0;

RTMeterCoordinates meterframe = new RTMeterCoordinates(startarcangle, arcextent,
           startarcscale, endarcscale, arcdirection, centerx, centery, arcradius);

meterframe.setGraphBorderDiagonal(0.025, 0.25, 0.175, 0.6) ;

RTMeterNeedleIndicator meterneedle = new RTMeterNeedleIndicator(meterframe, fuel);
meterneedle.setChartObjAttributes(attrib1);
meterneedle.setNeedleLength( 0.8);

ChartAttribute panelmeterattrib = new ChartAttribute(steelBlue,
    1,ChartConstants.LS_SOLID, Color.black);

.
.
.

chartVu.addChartObject(meterneedle);

RTMeterAxis meteraxis = new RTMeterAxis(meterframe, meterneedle);
meteraxis.setChartObjAttributes(attrib1);

meteraxis.setAxisTickDir(ChartConstants.AXIS_MIN);
meteraxis.setLineWidth( 2);
meteraxis.setColor( Color.white);
meteraxis.setAxisTickSpace(4);
meteraxis.setAxisMinorTicksPerMajor(4);
meteraxis.setShowAlarms( true);
meterneedle.setMeterAxis( meteraxis);
chartVu.addChartObject(meteraxis);

Font meterFont = font10Bold;

RTMeterStringAxisLabels meteraxislabels = new RTMeterStringAxisLabels(meteraxis);
meteraxislabels.setTextFont(meterFont);
meteraxislabels.setAxisLabelsDir(meteraxis.getAxisTickDir());
String [] labelstrings = {"E", "1/2", "F"};
meteraxislabels.setOverlapLabelMode( ChartConstants.OVERLAP_LABEL_DRAW);
meteraxislabels.setAxisLabelsEnds( ChartConstants.LABEL_MAX);
meteraxislabels.setAxisLabelsStrings(labelstrings,3);
meteraxislabels.setColor( Color.white);
chartVu.addChartObject(meteraxislabels);
```

9. Meter Indicators: Needle, Arc and Symbol

RTMeterIndicator
RTMeterArcIndicator
RTMeterNeedleIndicator
RTMeterSymbolIndicator

Familiar examples of analog meter indicators are voltmeters, car speedometers, pressure gauges, compasses and analog clock faces. Three meter indicator types are supported: arc, symbol, and needle meters. An unlimited number of meter indicators can be added to a given meter object. **RTPanelMeter** objects can be attached to an **RTMeterIndicator** object for the display of **RTProcessVar** numeric, alarm and string data in addition to the indicator graphical display. Meter scaling, meter axes, meter axis labels and alarm objects and handle by the meter coordinate system, meter axis and meter axis labels classes described in the preceding chapter.

Base Class for Meter Indicators

Class RTMeterIndicator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTMeterIndicator

The **RTMeterIndicator** class is the abstract base class for all meter indicators. Since it is abstract it does not have a constructor that you can use. It does have properties common to all meter indicator types and these are listed here.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAlarmIndicatorColorMode**(value) and value := **getAlarmIndicatorColorMode**();

AlarmIndicatorColorMode (inherited from **RTSingleValueIndicator**)	Get/Set whether the color of the indicator objects changes on an alarm. Use one of the constants: RT_INDICATOR_COLOR_NO_ALARM_CHANGE, RT_INDICATOR_COLOR_CHANGE_ON_ALARM..
ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a ChartAttribute object.
ChartObjEnable (inherited from **GraphObj**)	Enables/Disables the chart object. A chart object is drawn only if it is enabled. A chart object is enabled by default.
ChartObjScale (inherited from	Sets the reference to the **PhysicalCoordinates** object

GraphObj)	that the chart object is placed in
CurrentProcessValue (inherited from **RTSingleValueIndicator**)	Get the current process value of the primary channel.
FillColor (inherited from **GraphObj**)	Sets the fill color for the chart object.
IndicatorBackground	Get/Set the background attribute of the meter indicator.
IndicatorBackgroundEnable	Set to true to enable the display of the meter indicator background.
IndicatorSubtype	Set/Get the meter indicator subtype. Use one of the meter indicator subtype constants: RT_METER_NEEDLE_SIMPLE_SUBTYPE , RT_METER_NEEDLE_PIEWEDGE_SUBTYPE, RT_METER_NEEDLE_ARROW_SUBTYPE, RT_METER_ARC_BAND_SUBTYPE, RT_METER_SEGMENTED_ARC_SUBTYPE, RT_METER_SINGLE_SEGMENT_ARC_SUBTYPE, RT_METER_SYMBOL_ARC_SUBTYPE, RT_METER_SINGLE_SYMBOL_SUBTYPE.
LineColor (inherited from **GraphObj**)	Sets the line color for the chart object.
LineStyle (inherited from **GraphObj**)	Sets the line style for the chart object.
LineWidth (inherited from **GraphObj**)	Sets the line width for the chart object.
MeterAxis	Get/Set the reference meter axis.
NumChannels (inherited from **RTPlot**)	Get the number of channels in the indicator.
OverRangeNormalizedValue	Get/Set the displayable high end of the indicator range as a normalized value based on the **RTMeterCoordinates** and **RTMeterAxis** scale. For example, if the **RTMeterAxis** scale is 0 to 10, an overRangeNormalizedValue of 0.1 will allow the indicator to display off-scale up to 11.0.
PanelMeterList (inherited from **RTPlot**)	Set/Get the panel meter list of the **RTPlot** object.
PrimaryChannel (inherited from **RTPlot**)	Set/Get the primary channel of the indicator.
RTDataSource (inherited from **RTSingleValueIndicator**)	Get/Set the array list holding the **RTProcessVar** variables for the indicator.
UnderRangeNormalizedValue	Get/Set the displayable low end of the indicator range as a normalized value based on the **RTMeterCoordinates** and **RTMeterAxis** scale. For example, if the **RTMeterAxis** scale is 0 to 10, an underRangeNormalizedValue of -0.1 will allow the indicator to display off-scale down to -1.

ZOrder (inherited from **GraphObj**)	Sets the z-order of the object in the chart. Every object has a z-order value. Each z-order value does NOT have to be unique. If z-order sorting is turned on in the **ChartView** object, objects are sorted by z-order before they are drawn.

A complete listing of **RTMeterIndicator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Arc Meter Indicator

RTMeterArcIndicator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTMeterIndicator
RTMeterArcIndicator

This **RTMeterArcIndicator** class displays the current **RTProcessVar** value as an arc. Segmented meter arcs are one of the **RTMeterArcIndicator** subtypes. Varying the thickness of the arc, the segment width and segment spacing, and the segment end caps, will produce a wide variety of meter indicators. One of the advantages of the meter arc indicator is that it can be hollow in the center, allowing for the placement of a numeric panel meter as a digital readout in the center of the meter.

RTMeterArcIndicator constructor

```
public RTMeterArcIndicator(
   RTMeterCoordinates frame,
   RTProcessVar datasource,
   double innerarc,
   double outerarc,
   ChartAttribute attrib
);

public RTMeterArcIndicator(
   RTMeterCoordinates frame,
   RTProcessVar datasource,

);
```

Parameters

frame

The **RTMeterCoordinates** object defining the meter properties for the indicator.

datasource

The process variable associated with the indicator.

innerarc

The inner radius value in radius normalized units (0.0-1.0).

outerarc

The inner radius value in radius normalized units (0.0-1.0).

attrib

The color attributes of the indicator.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setIndicatorSubtype**(value) and value := **getIndicatorSubtype**();

IndicatorSubtype (inherited from **RTMeterIndicator**)	Set/Get the meter indicator subtype. Use one of the arc meter indicator subtype constants: RT_METER_ARC_BAND_SUBTYPE, RT_METER_SEGMENTED_ARC_SUBTYPE, RT_METER_SINGLE_SEGMENT_ARC_SUBTYPE.
InnerArcCapStyle	Set/Get the inner arc cap style. Use one of the constants:RT_METER_ARC_RADIUS_CAP, RT_METER_ARC_WEDGE_WIDTH_CAP, RT_METER_ARC_FLAT_CAP.
InnerValueArcNormalized	Set/Get the value of the inner arc radius in normalized radius coordinates.
OuterArcCapStyle	Set/Get the outer arc cap style. Use one of the constants:RT_METER_ARC_RADIUS_CAP, RT_METER_ARC_WEDGE_WIDTH_CAP, RT_METER_ARC_FLAT_CAP.
OuterValueArcNormalized	Set/Get the value of the outer arc radius in normalized radius coordinates.
SegmentSpacing	Set/Get the spacing of the arc segments in degrees.
SegmentValueRoundMode	Set/Get how the current process value is rounded up in calculating how many segments to display in RT_METER_SEGMENTED_ARC_SUBTYPE, RT_METER_SINGLE_SEGMENT_ARC_SUBTYPE modes. Use one of the constants: RT_FLOOR_VALUE, RT_CEILING_VALUE.
SegmentWidth	Set/Get the value of the arc segment width in degrees.

A complete listing of **RTMeterArcIndicator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

In the single segment arc indicator subtype (**RTMeterArcIndicator.setIndicatorSubType** =_METER_SINGLE_SEGMENT_ARC_SUBTYPE), only the last segment is “on”. The segments up to but not including the final segment are turned “off”,

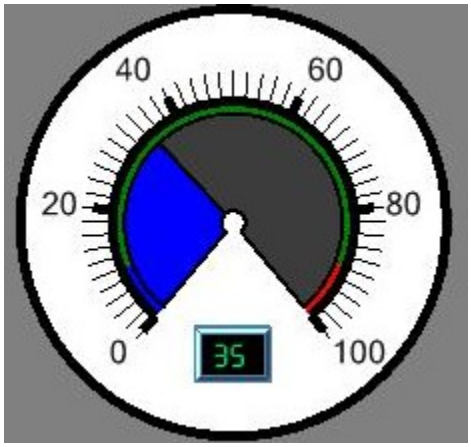
Examples for arc meter indicators

The examples below are program segments that give the important aspects of configuration an arc meter indicator for the image above it.

Extracted from the example program MeterDemo, file ArcMeter, method **InitializeMeter1**.

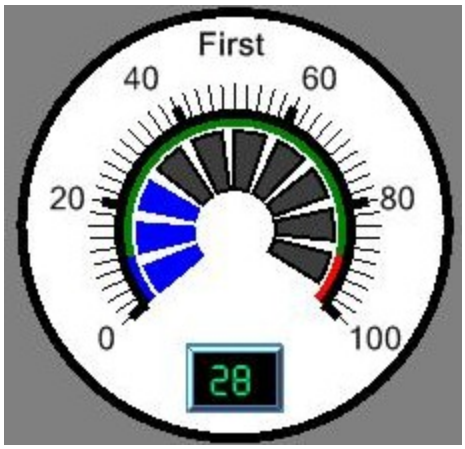

```
RTMeterArcIndicator meterarcindicator = new RTMeterArcIndicator(meterframe, processVar1);

meterarcindicator.setChartObjAttributes(attrib1);

meterarcindicator.setIndicatorSubtype (ChartConstants.RT_METER_ARC_BAND_SUBTYPE);

meterarcindicator.setInnerValueArcNormalized(0.65);

meterarcindicator.setOuterValueArcNormalized(0.85);

meterarcindicator.setIndicatorBackgroundEnable(true);

.

.

.


chartVu.addChartObject(meterarcindicator);
```

Extracted from the example program MeterDemo, file ArcMeter, method **InitializeMeter3**.



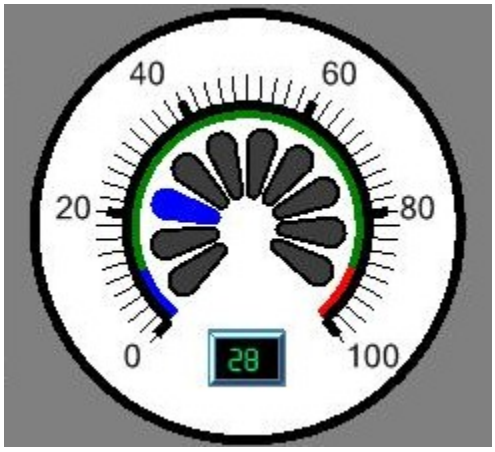

```
RTMeterArcIndicator meterarcindicator =
    new RTMeterArcIndicator(meterframe, processVar2);
meterarcindicator.setChartObjAttributes(attrib1);
meterarcindicator.setInnerValueArcNormalized(0.8);
meterarcindicator.setOuterValueArcNormalized(0.95);
meterarcindicator.setIndicatorSubtype(ChartConstants.RT_METER_ARC_BAND_SUBTYPE);
meterarcindicator.setIndicatorBackgroundEnable(true);
.
.
.
chartVu.addChartObject(meterarcindicator);
```

Extracted from the example program MeterDemo, file SegmentedArcMeter, method **InitializeMeter1**.




```
RTMeterArcIndicator meterarcindicator =
    new RTMeterArcIndicator(meterframe, processVar1);
meterarcindicator.setChartObjAttributes(attrib1);
meterarcindicator.setIndicatorSubtype(ChartConstants.RT_METER_SEGMENTED_ARC_SUBTYP
E);
meterarcindicator.setSegmentValueRoundMode(ChartConstants.RT_CEILING_VALUE);
meterarcindicator.setSegmentWidth(7);
meterarcindicator.setSegmentSpacing(10);
meterarcindicator.setInnerValueArcNormalized(0.35);
meterarcindicator.setOuterValueArcNormalized(0.85);
meterarcindicator.setIndicatorBackgroundEnable(true);
meterarcindicator.setIndicatorBackground( new ChartAttribute(Color.black,
    2,ChartConstants.LS_SOLID, new Color(60,60,60)));
.
.
.
chartVu.addChartObject(meterarcindicator);
```

Extracted from the example program MeterDemo, file SegmentedArcMeter, method **InitializeMeter3**.



```
RTMeterArcIndicator meterarcindicator =
    new RTMeterArcIndicator(meterframe, processVar2);
meterarcindicator.setChartObjAttributes(attrib1);
meterarcindicator.setInnerValueArcNormalized(0.8);
meterarcindicator.setOuterValueArcNormalized(0.95);

meterarcindicator.setIndicatorSubtype(ChartConstants.RT_METER_SEGMENTED_ARC_SUBTYP
E);
meterarcindicator.setSegmentValueRoundMode(ChartConstants.RT_CEILING_VALUE);
meterarcindicator.setSegmentWidth(7);
meterarcindicator.setSegmentSpacing(10);
meterarcindicator.setIndicatorBackgroundEnable(true);
.
.
chartVu.addChartObject(meterarcindicator);
```

Needle Meter Indicator

RTMeterNeedleIndicator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTMeterIndicator
RTMeterNeedleIndicator

This **RTMeterNeedleIndicator** class displays the current **RTProcessVar** value as a needle. Subtypes of the **RTMeterNeedleIndicator** are simple needles, pie wedge shaped needles (the fat end of the pie wedge is at the radius center) and arrow needles.

RTMeterNeedleIndicator Constructor

```
public RTMeterNeedleIndicator(
   RTMeterCoordinates frame,
   RTProcessVar datasource,
   double needlelength,
   double needleoverhang,
   double needlewidth,
   ChartAttribute attrib
);

public RTMeterNeedleIndicator(
   RTMeterCoordinates frame,
   RTProcessVar datasource,
);
```

Parameters

frame

The **RTMeterCoordinates** object defining the meter coordinate system.

datasource

The process variable associated with the meter indicator.

needlelength

Specifies length of the needle in normalized plot coordinates.

needleoverhang

Specifies the overhang of the back end of the needle indicator specified in needle radius normalized coordinates.

needlewidth

The color attributes of the meter indicator.

attrib

The color attributes of the meter indicator.

Selected Public Instance Properties*

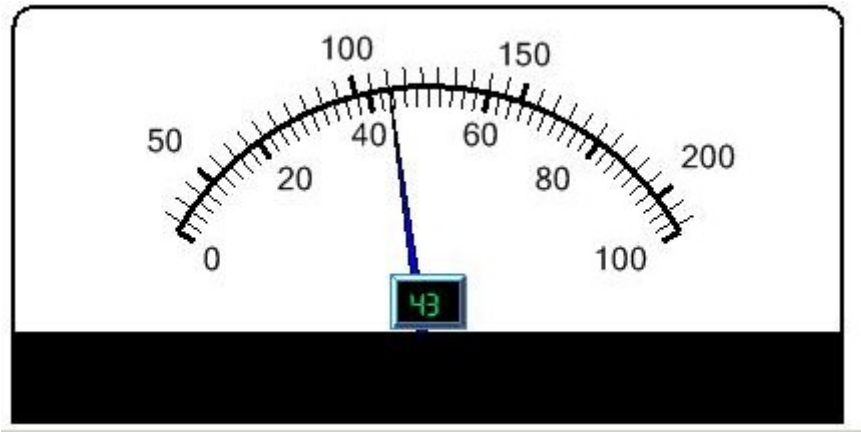
* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setIndicatorSubtype**(value) and value := **getIndicatorSubtype**();

IndicatorSubtype (inherited from **RTMeterIndicator**)	Set/Get the meter indicator subtype. Use one of the meter needle indicator subtype constants: RT_METER_NEEDLE_SIMPLE_SUBTYPE , RT_METER_NEEDLE_PIEWEDGE_SUBTYPE, RT_METER_NEEDLE_ARROW_SUBTYPE,
NeedleBaseWidth	Set/Get the width of the base end of the needle for the RT_METER_NEEDLE_SIMPLE_SUBTYPE needle type, in device coordinates.
NeedleHeadLengthMultiplier	Set/Get the head length multiplier for the RT_METER_NEEDLE_ARROW_SUBTYPE needle type, in device coordinates.
NeedleHeadWidthMultiplier	Set/Get the head width multiplier for the RT_METER_NEEDLE_ARROW_SUBTYPE needle type, in device coordinates.
NeedleLength	Set/Get the length of the needle in normalized plot coordinates. length.
NeedleOverhang	Set/Get the overhang of the back end of the needle indicator specified as a fraction of the needle length.
PieWedgeDegrees	Set/Get Specifies the arc width of the needle for the RT_METER_NEEDLE_PIEWEDGE_SUBTYPE needle type, in degrees coordinates.
PivotColor	Set/Get the color of the needle pivot.
PivotDrawFlag	Set to true to draw the needle pivot.
PivotRadius	Set/Get in device coordinates the radius of the pivot point of the needle, analogous to the bearing or axle supporting the meter needle.

A complete listing of **RTMeterNeedleIndicator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Examples for needle meter indicators

The examples below are program segments that give the important aspects of configuration a needle meter indicator for the image above it.



Extracted from the example program MeterDemo, file NeedleMeter, method **InitializeMeter2**.

```
RTMeterNeedleIndicator meterneedle =
    new RTMeterNeedleIndicator(meterframe1, processVar2);
meterneedle.setChartObjAttributes(attrib1);
meterneedle.setNeedleLength(0.7);
meterneedle.setZOrder(55);

ChartAttribute panelmeterattrib = new ChartAttribute(Color.lightGray,
    3,ChartConstants.LS_SOLID, Color.black);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(meterframe1,
     processVar1,panelmeterattrib);
panelmeter.setPanelMeterPosition(ChartConstants.CUSTOM_POSITION);
panelmeter.getNumericTemplate().setXJust(ChartConstants.JUSTIFY_CENTER);
panelmeter.getNumericTemplate().setYJust(ChartConstants.JUSTIFY_MIN);

panelmeter.setLocation(0.0, 0.2);
panelmeter.getNumericTemplate().setTextFont(font16Numeric);
meterneedle.addPanelMeter(panelmeter);

ChartAttribute panelmetertagattrib = new ChartAttribute(Color.lightGray,
    3,ChartConstants.LS_SOLID, Color.white);
RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(meterframe1,
    processVar1, panelmetertagattrib, ChartConstants.RT_TAG_STRING);
panelmeter3.setPanelMeterPosition(ChartConstants.BELOW_REFERENCED_TEXT);
panelmeter3.setPositionReference( panelmeter);
panelmeter3.setPanelMeterNudge(0,0);
panelmeter3.getStringTemplate().setXJust(ChartConstants.JUSTIFY_CENTER);
panelmeter3.getStringTemplate().setYJust(ChartConstants.JUSTIFY_MAX);
panelmeter3.setTextColor(Color.black);
panelmeter3.getStringTemplate().setTextFont(font12Bold);
meterneedle.addPanelMeter(panelmeter3);

chartVu.addChartObject(meterneedle);
```

Extracted from the example program MeterDemo, file ArrowMeter, method **InitializeMeter8**.


```
RTMeterNeedleIndicator meterneedle =
    new RTMeterNeedleIndicator(meterframe, processVar1);
meterneedle.setNeedleLength(0.6);
meterneedle.setChartObjAttributes(attrib1);

ChartAttribute panelmeterattrib = new ChartAttribute(Color.lightGray,
    3, ChartConstants.LS_SOLID, Color.black);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(meterframe,
    processVar1,panelmeterattrib);
panelmeter.setPanelMeterPosition(ChartConstants.RADIUS_TOP);
panelmeter.getNumericTemplate().setXJust(ChartConstants.JUSTIFY_MAX);
panelmeter.getNumericTemplate().setYJust(ChartConstants.JUSTIFY_CENTER);
panelmeter.setPanelMeterNudge(-16,0);
panelmeter.getNumericTemplate().setTextFont(font16Numeric);
meterneedle.setIndicatorSubtype(ChartConstants.RT_METER_NEEDLE_ARROW_SUBTYPE);
meterneedle.addPanelMeter(panelmeter);

chartVu.addChartObject(meterneedle);
```

Symbol Meter Indicators

Class RTMeterSymbolIndicator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTMeterIndicator
RTMeterSymbolIndicator

This **RTMeterSymbolIndicator** class displays the current **RTProcessVar** value as a symbol moving around in the meter arc. Symbols include all of the **QCChart2D** scatter plot symbols: SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, BAR3D, and CIRCLE.

RTMeterSymbolIndicator constructor

```
public RTMeterSymbolIndicator(
   RTMeterCoordinates frame,
   RTProcessVar datasource,
   int symbolnum,
   double symbolsize,
   ChartAttribute attrib
);
public RTMeterSymbolIndicator(
   RTMeterCoordinates frame,
   RTProcessVar datasource,

);
```

Parameters

frame

The **RTMeterCoordinates** object defining the meter properties for the indicator.

datasource

The process variable associated with the indicator.

symbolnum

Specifies what symbol to use in the indicator. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, BAR3D, CIRCLE.

symbolsize

The size of the symbol in points.

attrib

The color attributes of the indicator.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setIndicatorSubtype**(value) and value := **getIndicatorSubtype**();

IndicatorSubtype (inherited from **RTMeterIndicator**)	Set/Get the meter indicator subtype. Use one of the meter symbol indicator subtype constants:, RT_METER_SYMBOL_ARC_SUBTYPE, RT_METER_SINGLE_SYMBOL_SUBTYPE.
SegmentValueRoundMode	Get/Set that the current process value is rounded up in calculating how many symbols to display in RT_METER_SYMBOL_ARC_SUBTYPE, RT_METER_SINGLE_SYMBOL_SUBTYPE modes. Use one of the constants: RT_FLOOR_VALUE, RT_CEILING_VALUE.
SymbolNum	Set/Get the symbol used as the indicator symbol. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR,

	BAR3D, CIRCLE.
SymbolPosPercent	Set/Get the radial position of the symbol indicator.
SymbolSize	Set/Get the size of the symbol indicator in points.
SymbolSpacing	Get/Set the space, in degrees, between adjacent symbols.

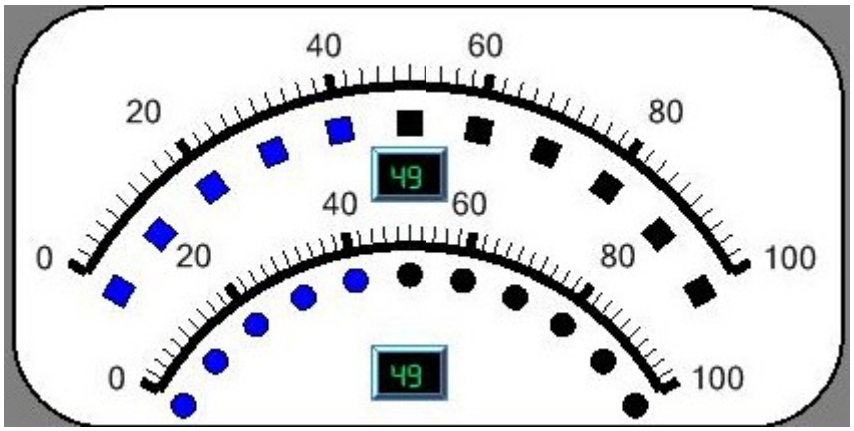
A complete listing of **RTMeterSymbolIndicator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

In the single symbol indicator subtype (**RTMeterSymbolIndicator.setIndicatorSubType** =_ RT_METER_SINGLE_SYMBOL_SUBTYPE), only the last symbol is "on". The symbols up to but not including the final symbol are turned "off",

Examples for symbol meter indicators

The examples below are program segments that give the important aspects of configuration a needle meter indicator for the image above it.

The top meter indicator, extracted from the example program MeterDemo, file SymbolMeter, method **InitializeMeter2**.




```
RTMeterSymbolIndicator metersymbolindicator =

        new RTMeterSymbolIndicator(meterframe, processVar2);

metersymbolindicator.setChartObjAttributes(attrib1)

metersymbolindicator.setIndicatorSubtype(ChartConstants.RT_METER_SYMBOL_ARC_SUBTYPE);
```

```
metersymbolindicator.setSymbolSpacing(10);

metersymbolindicator.setSymbolNum(ChartConstants.SQUARE);

metersymbolindicator.setIndicatorBackgroundEnable(true);

metersymbolindicator.setSymbolSize(12);

metersymbolindicator.setSymbolPosPercent(0.9);

ChartAttribute panelmeterattrib = new ChartAttribute(Color.lightGray,3,

        ChartConstants.LS_SOLID, Color.black);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(meterframe,

        processVar1, panelmeterattrib);

panelmeter.setPanelMeterPosition(ChartConstants.CUSTOM_POSITION);

panelmeter.setLocation(0.0, 0.7);

panelmeter.getNumericTemplate().setXJust(ChartConstants.JUSTIFY_CENTER);

panelmeter.getNumericTemplate().setYJust(ChartConstants.JUSTIFY_MIN);

panelmeter.getNumericTemplate().setTextFont(font16Numeric);

metersymbolindicator.addPanelMeter(panelmeter);



chartVu.addChartObject(metersymbolindicator);
```

The bottom meter indicator, extracted from the example program MeterDemo, file SymbolMeter, method **InitializeMeter3**.

```
RTMeterSymbolIndicator metersymbolindicator =
    new RTMeterSymbolIndicator(meterframe, processVar2);
metersymbolindicator.setChartObjAttributes(attrib1);
metersymbolindicator.setIndicatorSubtype(ChartConstants.RT_METER_SYMBOL_ARC_SUBTYP
E);
metersymbolindicator.setSymbolSpacing(10);
metersymbolindicator.setSymbolNum(ChartConstants.CIRCLE);
metersymbolindicator.setIndicatorBackgroundEnable(true);
metersymbolindicator.setSymbolSize(12);
metersymbolindicator.setSymbolPosPercent(0.9);

ChartAttribute panelmeterattrib = new ChartAttribute(Color.lightGray,3,
    ChartConstants.LS_SOLID, Color.black);

RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(meterframe,
    processVar1, panelmeterattrib);
panelmeter.setPanelMeterPosition(ChartConstants.CUSTOM_POSITION);
panelmeter.setLocation(0.0, 0.2);
panelmeter.getNumericTemplate().setXJust(ChartConstants.JUSTIFY_CENTER);
panelmeter.getNumericTemplate().setYJust(ChartConstants.JUSTIFY_MIN);
```

```
panelmeter.getNumericTemplate().setTextFont(font16Numeric);
metersymbolindicator.addPanelMeter(panelmeter);

chartVu.addChartObject(metersymbolindicator);
```

10. Dials and Clocks

RTComboProcessVar
RTMeterNeedleIndicator

Clocks and dials use the same meter components as described in the previous chapter: **RTMeterCoordinates, RTMeterAxis, RTMeterAxisLabels, RTMeterStringAxisLabels, RTMeterIndicator, RTMeterArcIndicator, RTMeterNeedleIndicator,** and **RTMeterSymbolIndicator**. For the purposes of this discussion, a dial and a clock is a meter that has an arc extent of 360 degrees, i.e. a full circle. Also, the current value of the dial or clock is characterized by a single value. In the case of a clock it is a date/time value and in the case of a dial it is a simple numeric value. While they are characterized by a single value, dials and clocks have multiple meter indicators representing that value using varying degrees of precision. Everyone is familiar with the hour, minute and second hands of clocks so we don't need to describe that further. As for a dial, the aircraft altimeter gauge is a perfect example. It has a small hand representing thousands of feet and a large hand representing hundreds. Clocks and dials must be able to take a single value, either time or some floating point value, and translate that information into two or more meter indicator values. In the case of a clock, the current time must be converted into values for the hour, minute and second hands. The class responsible for this conversion is the **RTComboProcessVar** class. It converts a single **RTProcessVar** value into multiple **RTProcessVar** objects, one representing the current value of each indicator of the clock or dial.

Converting Dial and Clock Data using RTComboProcessVar

Class RTComboProcessVar

RTProcessVar
    RTComboProcessVar

The **RTComboProcessVar** class has an internal collection of **RTProcessVar** objects. The current value assigned to the **RTComboProcessVar** object is simultaneously converted to current values for each of the **RTProcessVar** objects in the collection. For each **RTProcessVar** object, the conversion is defined by a divisor and a modulo N value. Each **RTProcssVar** object will have unique combination of divisors and modulo N values as defining characteristics.

```
For i=0 to  processVarList.Count-1
```

```
processVarList[i].setCurrentValue(
    (comboProcessVar.getCurrentValue() / divisor[i] ) % modvalue[i])
```

where

comboProcessVar

The main **RTComboProcessVar** object that is updated by the application program

processVarList

The collection of **RTProcessVar** objects internal to the **RTComboProcessVar**. These items are updated automatically by the master **RTComboProcessVar** whenever an update is made to the master class.

Note that the divisor operation takes place first, followed by the modulo operation.

RTComboProcessVar constructors

```
public RTComboProcessVar(
   TimeSimpleDataset dataset,
   ChartAttribute defaultattribute
);
```

```
public RTComboProcessVar(
   string tagname,
   ChartAttribute defaultattribute
);
```

Parameters

dataset
A dataset that will be used to store historical values.

defaultattribute
Specifies the default attribute for this process variable.

tagname
The string representing the tag name of the process variable.

Selected Public Instance Methods

addProcessVar	Adds a new process variable to the process variable list.
resetProcessVarsList	Clears the process variable list.

setCurrentValue	Overloaded. Updates the current value and the dataset of the underlying **RTProcessVar**. It also updates the process variable list with the calculated process values.
setDivisorItem	Sets the divisor factor at the specified index.
setModuloItem	Sets the modulo factor at the specified index.

A complete listing of **RTComboProcessVar** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Examples for using RTComboProcessVar in a clock application

The example, extracted from the example program AutoInstrumentPanel, methods **InitializeGraph** and **InitializeClock**, show the important aspects of using an **RTComboProcessVar** object to supply data for the three meter needle indicators used as the hands of a clock.


```
RTProcessVar [] clockdata = new RTProcessVar[3];
RTComboProcessVar clock12Hour;  // 12-hour clock.
.
.
clockdata[0] = new RTProcessVar("Seconds", defaultattrib);
clockdata[1] = new RTProcessVar("Minutes", defaultattrib);
clockdata[2] = new RTProcessVar("Hours", defaultattrib);

clock12Hour = new  RTComboProcessVar("12-Hour Clock", defaultattrib);
clock12Hour.addProcessVar(clockdata[0]); // seconds
clock12Hour.addProcessVar(clockdata[1]); // minutes
clock12Hour.addProcessVar(clockdata[2]); // hours
// Clock/Meter coordinates is going to be scaled from 0-12,
// All values must be converted to this range
clock12Hour.setDivisorItem(0,5); // seconds/5 give seconds position on 0-12 scale
clock12Hour.setDivisorItem(1,5*60); // seconds/300 give minutes on 0-12 scale
clock12Hour.setDivisorItem(2,60 * 60); // seconds / 3600 give hours on 0-12 scale
clock12Hour.setModuloItem(0,12); // apply modulo 12 base
clock12Hour.setModuloItem(1,12); // apply modulo 12 base
clock12Hour.setModuloItem(2,12); // apply modulo 12 base.
.
.
private void InitializeClock()
{
    ChartView chartVu  = this;
    double startarcangle = 90;
    double arcextent = 360;
    double startarcscale = 0.0;
    double endarcscale = 12.0;
```

```
    boolean arcdirection = false;
    double arcradius = 0.50;
    double centerx = 0.0, centery= -0.0;
    Font meterFont = font12;
    RTMeterCoordinates meterframe =
        new RTMeterCoordinates(startarcangle, arcextent,
    startarcscale, endarcscale, arcdirection, centerx, centery, arcradius);

    meterframe.setGraphBorderDiagonal(0.8, .0, 0.99, 0.3) ;

    ChartAttribute frameattrib = new ChartAttribute (Color.black,
        3,ChartConstants.LS_SOLID, Color.blue);

            // Seconds
    ChartAttribute needleattrib1 = new ChartAttribute (Color.black,
        1,ChartConstants.LS_SOLID, Color.blue);
    RTMeterNeedleIndicator meterneedle1 =
        new RTMeterNeedleIndicator(meterframe, clockdata[0]);
    meterneedle1.setNeedleBaseWidth( 1);
    meterneedle1.setChartObjAttributes(needleattrib1);
    meterneedle1.setNeedleLength( 0.5);
    chartVu.addChartObject(meterneedle1);

    // Minutes
    ChartAttribute needleattrib2 = new ChartAttribute (Color.black,
        1,ChartConstants.LS_SOLID, Color.blue);
    RTMeterNeedleIndicator meterneedle2 =
        new RTMeterNeedleIndicator(meterframe, clockdata[1]);
    meterneedle2.setNeedleBaseWidth( 3);
    meterneedle2.setChartObjAttributes(needleattrib2);
    meterneedle2.setNeedleLength( 0.45);
    chartVu.addChartObject(meterneedle2);

            // Hours
    ChartAttribute needleattrib3 = new ChartAttribute (Color.black,
         1,ChartConstants.LS_SOLID, Color.blue);
    RTMeterNeedleIndicator meterneedle3 = new RTMeterNeedleIndicator(meterframe,
        clockdata[2]);
    meterneedle3.setNeedleBaseWidth( 5);
    meterneedle3.setChartObjAttributes(needleattrib3);
    meterneedle3.setNeedleLength( 0.3);
    chartVu.addChartObject(meterneedle3);

.
.
.

}
```

Examples for using RTComboProcessVar in an altimeter application

The example, extracted from the example program MeterDemo, file DialsAndClocks, methods **InitializeGraph** and **InitializeDial1**, show the important aspects of using an **RTComboProcessVar** object to supply data for the two meter needle indicators used as the hands of a clock.


```
dialComboProcessVar1 = new  RTComboProcessVar("Altimeter",
    processVar1.getDefaultAttribute());
dialComboProcessVar1.addProcessVar(bigProcessVarArray[0]);
dialComboProcessVar1.addProcessVar(bigProcessVarArray[1]);

dialComboProcessVar1.setDivisorItem(0,1);
dialComboProcessVar1.setDivisorItem(1,10);

dialComboProcessVar1.setModuloItem(0,100);
dialComboProcessVar1.setModuloItem(1,100);.
.
.
private void InitializeDial1()
{

    ChartView chartVu  = this;
    double startarcangle = 90;
    double arcextent = 360;
    double startarcscale = 0.0;
    double endarcscale = 100.0;
    boolean arcdirection = false;
    double arcradius = 0.90;
    double centerx = 0.0, centery= 0.0;
    Font meterFont =font12;

    RTMeterCoordinates meterframe = new RTMeterCoordinates(startarcangle,
        arcextent, startarcscale, endarcscale, arcdirection, centerx,
        centery, arcradius);

    meterframe.setGraphBorderDiagonal(0.0, 0.0, 0.25, 0.45) ;

    Background gbackground = new Background( meterframe,
        ChartConstants.GRAPH_BACKGROUND, Color.white);
    chartVu.addChartObject(gbackground);

    ChartAttribute frameattrib = new ChartAttribute (Color.black,
        3,ChartConstants.LS_SOLID, Color.blue);

    ChartAttribute needleattrib1 = new ChartAttribute (Color.black,
        1,ChartConstants.LS_SOLID, Color.blue);
    RTMeterNeedleIndicator meterneedle1 = new RTMeterNeedleIndicator(meterframe,
         bigProcessVarArray[0]);
    meterneedle1.setNeedleBaseWidth(5);
    meterneedle1.setChartObjAttributes(needleattrib1);
    meterneedle1.setNeedleLength(0.55);
    chartVu.addChartObject(meterneedle1);

    ChartAttribute needleattrib2 = new ChartAttribute (Color.black,
        1,ChartConstants.LS_SOLID, Color.blue);
```

```
    RTMeterNeedleIndicator meterneedle2 = new RTMeterNeedleIndicator(meterframe,
        bigProcessVarArray[1]);
    meterneedle2.setNeedleBaseWidth(3);
    meterneedle2.setChartObjAttributes(needleattrib2);
    meterneedle2.setNeedleLength(0.35);
    chartVu.addChartObject(meterneedle2);
.
.
.

}

.
.
.
End Sub 'InitializeDial1
```

11. Single and Multiple Channel Annunciators

RTAnnunciator
RTMultiValueAnnunciator

An annunciator is used to display the current values and alarm states of real-time data. Each channel of data corresponds to a rectangular cell where each cell can contain the tag name, units, current value, and alarm status message. Any of these items may be excluded. If a channel is in alarm, the background of the corresponding cell changes its color, giving a strong visual indication that an alarm has occurred.

Single Channel Annunciator

Class RTAnnunciator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTMultiValueIndicator
RTAnnunciator

An **RTAnnunciator** is used to display the current values and alarm states of a single channel real-time data.

RTAnnunciator constructor

```
public RTAnnunciator(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   ChartRectangle2D annunpos,
   ChartAttribute attrib
);
```

Parameters

transform
The coordinate system for the new **RTAnnunciator** object.
datasource
The process variable associated with the annunciator.
annunpos
The position and size of the annunciator.
attrib
The color attributes of the annunciator.

The annunciator resides in coordinate system scaled for physical coordinates of (0.0,0.0) – (1.0, 1.0). The annunciator rectangle size and position is defined using the **RTAnnunciator.AnnunciatorRect** property. The default annunciator consists of a simple rectangle that changes color in response to the alarm state of the **RTProcessVar** object attached to the annunciator. The annunciator can be customized with tag names, numeric readouts and alarm messages by adding **RTPanelMeter** objects to the **RTAnnunciator** object. See the examples below.

Public Instance Properties

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAnnunciatorRect**(value) and value := **getAnnunciatorRect**();

AnnunciatorRect	Get/Set the annunciator rectangle.

A complete listing of **RTAnnuciator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for single channel annunciator

The example below, extracted from the AnnunciatorDemo example, file Annunciators, method **InitializeAnnunciator1**, creates a single channel annunciator with a tag name, numeric readout and alarm.


```
private void InitializeAnnunciator1()
{
    ChartView chartVu  = this;

    CartesianCoordinates pTransform1 =
        new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
    pTransform1.setGraphBorderDiagonal(0.01, .05, 0.15, 0.35) ;
    Background background = new Background( pTransform1,
        ChartConstants.PLOT_BACKGROUND, Color.gray);
    chartVu.addChartObject(background);
```

```
    ChartRectangle2D annunrect = new ChartRectangle2D(0.05, 0.05, 0.9, 0.9);
    ChartAttribute attrib1 = new ChartAttribute (Color.darkGray,
        5,ChartConstants.LS_SOLID, Color.darkGray);
    RTAnnunciator annunciator = new RTAnnunciator(pTransform1, processVar1,
        annunrect, attrib1);

    ChartAttribute panelmeterattrib = new ChartAttribute(Color.lightGray,3,
        ChartConstants.LS_SOLID, Color.black);
    RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(pTransform1,
        processVar2,panelmeterattrib);
    panelmeter.setPanelMeterPosition(ChartConstants.PLOTAREA_CENTER);
    panelmeter.getNumericTemplate().setTextFont(font24Numeric);
    panelmeter.getNumericTemplate().setPostfixString( ((char) 176) + "F");
    annunciator.addPanelMeter(panelmeter);

    RTAlarmPanelMeter panelmeter2 = new RTAlarmPanelMeter(pTransform1,
        processVar1,panelmeterattrib);
    panelmeter2.setPanelMeterPosition(ChartConstants.INSIDE_BARBASE);
    panelmeter2.getAlarmTemplate().setTextFont(font12);
    panelmeter2.setPositionReference( panelmeter);
    annunciator.addPanelMeter(panelmeter2);

    ChartAttribute panelmetertagattrib = new ChartAttribute(Color.lightGray,3,
        ChartConstants.LS_SOLID, Color.white);
    RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(pTransform1,
        processVar1, panelmetertagattrib, ChartConstants.RT_TAG_STRING);
    panelmeter3.setPositionReference( panelmeter);
    panelmeter3.setPanelMeterPosition(ChartConstants.INSIDE_BAR);
    panelmeter3.setTextColor(Color.black);
    annunciator.addPanelMeter(panelmeter3);


    chartVu.addChartObject(annunciator);
}
```

Multi-Channel Annunciators

Class RTMultiValueAnnunciator

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTMultiValueIndicator
RTMultiValueAnnunciator

An **RTMultiValueAnnunciator** is used to display the current values and alarm states of a collection of **RTProcessVar** objects. It consists of a rectangular grid with individual channels represented by the rows and columns in of the grid. Each grid cell can contain the tag name, units, current value, and alarm status message for a single **RTProcessVar** object. Any of these items may be excluded. If a channel is in alarm, the background of the corresponding cell changes its color, giving a strong visual indication that an alarm has occurred.

RTMultiValueAnnunciator constructors

```
public RTMultiValueAnnunciator(
```

```
    PhysicalCoordinates transform,
    RTProcessVar[] datasource,
    int numcols,
    int numrows,
    ChartAttribute[] attribs
);
```

Parameters

transform

The coordinate system for the new **RTMultiValueAnnunciator** object.

datasource

An array of **RTProcessVar** objects, one for each annunciator cell.

numcols

The number of columns in the annunciator display.

numrows

The number of rows in the annunciator display.

attribs

An array of the color attributes one for each annunciator cell.

Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setCellColumnMargin**(value) and value := **getCellColumnMargin**();

CellColumnMargin	Get/Set the extra space between columns of the annunciator, specified in normalized NORM_PLOT_POS coordinates.
CellRowMargin	Get/Set the extra space between rows of the annunciator, specified in normalized NORM_PLOT_POS coordinates.
NumberColumns	Get the number of rows in the annunciator.
NumberRows	Get the number of rows in the annunciator.

A complete listing of **RTMultiValueAnnunciator** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

The **CellColumnMargin** and the **CellRowMargin** values represent the total amount of spacing used for the cell column and cell row margins respectively. A value of 0.2 implies that 20% of the row or column space will be used as margin, and 80% will be used for the annunciator cells. The 20% margin value is divided up between the cells in the row or column. If the multi-channel annunciator has 4 annunciator cells in a row, there are 5 border areas between the cells (3 at the interior of the annunciator cell grid and 2 on either end). The total margin of 20% is therefore divided 5 times, resulting in a 4% margin between the column of each grid cell.

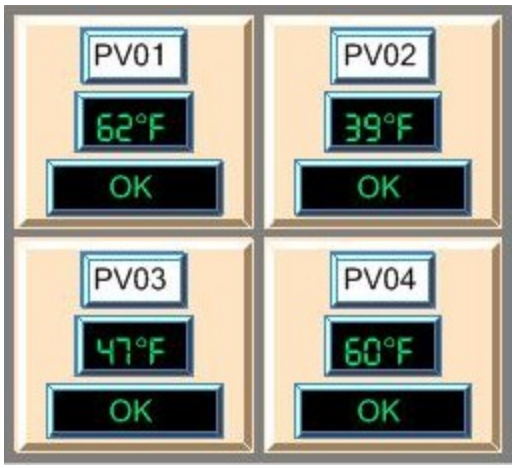
Example for a simple multi-channel annunciator

The example below, extracted from the AutoInstrumentPanel example, method **InitializeAnnunciator**, creates a multi-channel annunciator that shows only the tag name of the associated **RTProcessVar** object.


```
private void InitializeAnnunciator()
{

    ChartView chartVu  = this;
    CartesianCoordinates pTransform1 =
        new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
    pTransform1.setGraphBorderDiagonal(0.05, 0.1, 0.725, 0.175) ;

    ChartAttribute attrib;
    ChartAttribute []attribArray = new ChartAttribute[annunciator1.length];

    for (int i=0; i < annunciator1.length; i++)
    {
      attrib = new ChartAttribute (darkRed, 3,ChartConstants.LS_SOLID, darkRed);

      attribArray[i] = attrib;
    }

    int numrows = 1;
    int numcols = 8;
    RTMultiValueAnnunciator annunciator =
        new RTMultiValueAnnunciator(pTransform1, annunciator1, numcols,
        numrows, attribArray);
    annunciator.setCellRowMargin( 0.05);
    annunciator.setCellRowMargin( 0.1);

    ChartAttribute panelmetertagattrib = new ChartAttribute(steelBlue,3,
        ChartConstants.LS_SOLID, Color.white);
    RTStringPanelMeter panelmeter = new RTStringPanelMeter(pTransform1,
        annunciator1[0], panelmetertagattrib, ChartConstants.RT_TAG_STRING);
    panelmeter.getStringTemplate().setTextFont(font10Bold);
    panelmeter.setPanelMeterPosition( ChartConstants.INSIDE_BAR);
    panelmeter.setFrame3DEnable( false);
    panelmeter.setTextColor( Color.white);
    panelmeter.getStringTemplate().setTextBgMode( false);
    panelmeter.setAlarmIndicatorColorMode(
        ChartConstants.RT_INDICATOR_COLOR_NO_ALARM_CHANGE);
    annunciator.addPanelMeter(panelmeter);
    chartVu.addChartObject(annunciator);
}
```

Example for a simple multi-channel annunciator

The example below, extracted from the AnnunciatorDemo example, file Annunciators, method **InitializeAnnunciator2**, creates a multi-channel annunciator that shows the tag name, current value and alarm state of the associated **RTProcessVar** object.

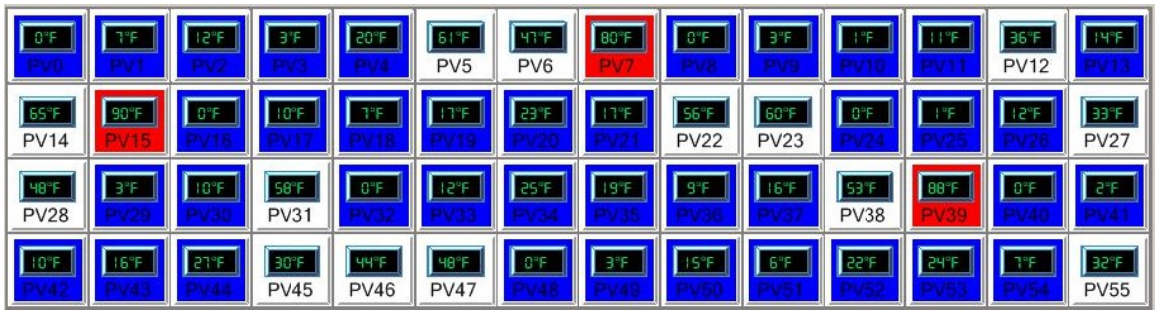



```
private void InitializeAnnunciator2()
{
    RTProcessVar [] processVarArray = {processVar1, processVar2,
        processVar3, processVar4};

    ChartView chartVu  = this;

    CartesianCoordinates pTransform1 =
        new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
    pTransform1.setGraphBorderDiagonal(0.175, .005, 0.45, 0.475) ;

    Background background = new Background( pTransform1,
        ChartConstants.PLOT_BACKGROUND, Color.gray);
    chartVu.addChartObject(background);

    Color bisque = new Color(0xFFE4C4);
    ChartAttribute attrib1 = new ChartAttribute (bisque,
        5,ChartConstants.LS_SOLID, bisque);
    ChartAttribute attrib2 = new ChartAttribute (bisque,
        5,ChartConstants.LS_SOLID, bisque);
    ChartAttribute attrib3 = new ChartAttribute (bisque,
        5,ChartConstants.LS_SOLID, bisque);
    ChartAttribute attrib4 = new ChartAttribute (bisque,
        5,ChartConstants.LS_SOLID, bisque);

    ChartAttribute []attribArray = {attrib1, attrib2, attrib3, attrib4};

    int numrows = 2;
    int numcols = 2;
    RTMultiValueAnnunciator annunciator =
         new RTMultiValueAnnunciator(pTransform1, processVarArray,
            numcols, numrows, attribArray);

    ChartAttribute panelmeterattrib = new ChartAttribute(Color.lightGray,3,
        ChartConstants.LS_SOLID, Color.black);
    RTNumericPanelMeter panelmeter = new RTNumericPanelMeter(pTransform1,
        processVar2,panelmeterattrib);
    panelmeter.setPanelMeterPosition(ChartConstants.CENTERED_BAR);
    panelmeter.getNumericTemplate().setTextFont(font24Numeric);
    panelmeter.getNumericTemplate().setPostfixString( ((char) 176) + "F");
    annunciator.addPanelMeter(panelmeter);

    RTAlarmPanelMeter panelmeter2 = new RTAlarmPanelMeter(pTransform1,
        processVar1,panelmeterattrib);

    panelmeter2.setPanelMeterPosition(ChartConstants.BELOW_REFERENCED_TEXT);
    panelmeter2.getAlarmTemplate().setTextFont(font12);
    panelmeter2.setPositionReference( panelmeter);
```

```
    annunciator.addPanelMeter(panelmeter2);

    ChartAttribute panelmetertagattrib = new
        ChartAttribute(Color.lightGray,3,ChartConstants.LS_SOLID, Color.white);
    RTStringPanelMeter panelmeter3 = new RTStringPanelMeter(pTransform1,
        processVar1, panelmetertagattrib, ChartConstants.RT_TAG_STRING);
    panelmeter3.setPositionReference(panelmeter);
    panelmeter3.getStringTemplate().setTextFont(font12);

    panelmeter3.setPanelMeterPosition(ChartConstants.ABOVE_REFERENCED_TEXT);
    panelmeter3.setTextColor(Color.black);
    annunciator.addPanelMeter(panelmeter3);

    chartVu.addChartObject(annunciator);
}
```

Example for a large multi-channel annunciator

The example below, extracted from the AnnunciatorDemo, file Annunciators, method **InitializeAnnunciator4** example, creates a multi-channel annunciator that shows the tag name and current value of the associated **RTProcessVar** object. The alarm state is implicit in the annunciator background color. See the example program for the code listing.



12. The Scroll Frame and Single Channel Scrolling Plots

RTScrollFrame
RTSimpleSingleValuePlot

Scrolling graphs are built using three main. The first is the RTScrollFrame class that manages the constant rescaling of the coordinate system of the scrolling graph. The second and third are **RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot** classes that encapsulate the actual line plot, bar plot, scatter plot or group plot that is plotted in the scrolling graph. The **RTScrollFrame** class and the **RTSimpleSingleValuePlot** classes are described in this chapter and the **RTGroupMultiValuePlot** class is described in the next.

The original **RTScrollFrame** manges scrolling of numeric, time/date and elapsed time coordinate systems in the horizontal direction. Starting with Revision 2.0, a new scroll frame has been added, **RTVerticalScrollFrame**, which manages scrolling in the vertical direction.

Scroll Frame

Class RTScrollFrame

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTMultiValueIndicator
RTScrollFrame

The scrolling algorithm used in this software is different that in earlier Quinn-Curtis real-time graphics products. Scrolling plots are no longer updated incrementally whenever the underlying data is updated. Instead, the underlying **RTProcessVar** data objects are updated as fast as you want. Scrolling graphs (all graphs for that matter) are only updated with the ChartView.updateDraw() method is called. What makes scrolling graphs appear to scroll is the scroll frame (**RTScrollFrame**). When a scroll frame is updated as a result of the ChartView.updateDraw() event, it analyzes the **RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot** objects that have been attached to it and creates a coordinate system that matches the current and historical data associated with the plot objects. The plot objects in the scroll frame are drawn into this coordinate system. As data progresses

forward in time the coordinate system is constantly being rescaled to include the most recent time values as part of the x-coordinate system. You can control whether or not the starting point of the scroll frame coordinate system remains fixed, whether it advances in sync with the constantly changing end of the scroll frame. Other options allow the y-scale to be constantly rescaled to reflect the current dynamic range of the y-values in the scroll frame. The long term goal is that as computers get faster, and Java more efficient, you will never need to update the display faster than 30-60 times a second, since this will result smooth scrolling even if the underlying data is updated 10,000 times a second. Computers are not there yet (running Java at least; other languages are much faster) and the scrolling may appear slow, but in the long run the algorithm should prove an efficient technique throughout the rest of the decade.

RTScrollFrame constructors

```
public RTScrollFrame(
   ChartView component,
   RTProcessVar processvar,
   PhysicalCoordinates initialscale,
   int scrollxmode,
   int autoscaleymode
);
public RTScrollFrame(
   ChartView component,
   RTProcessVar processvar,
   PhysicalCoordinates initialscale,
   int scrollmode
);

public RTScrollFrame(
   ChartView component,
   PhysicalCoordinates initialscale,
   int scrollxmode,
   int autoscaleymode
);
```

Parameters

component

This **ChartView** component the scroll frame is placed in.

processvar

The source process variable.

initialscale

A coordinate system that serves as the initial scale for the scroll frame.

scrollxmode

Specifies x-axis auto-scale mode of the scroll frame. Use one of the x-axis scroll frame constants:
RT_NO_AUTOSCALE_X - no auto-scale for the x-axis, use in non-scrolling graphs.
RT_AUTOSCALE_X_CURRENT_SCALE - auto-scale based on current scale, use in non-scrolling graphs.

RT_AUTOSCALE_X_MIN - autoscale x-axis minimum only, use in non-scrolling graphs.
RT_AUTOSCALE_X_MAX - autoscale x-axis maximum only, use in non-scrolling graphs.
RT_AUTOSCALE_X_MINMAX - autoscale x-axis minimum and maximum, use in non-scrolling graphs.
RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL - autoscale the x-axis for a fixed range, with moving maximum and minimum values, use in scrolling graphs.
RT_MAXEXTENT_FIXEDSTART_AUTOSCROLL - autoscale the x-axis with the start of the x-axis fixed, and the end of the x-axis moving, use in scrolling graphs.
RT_FIXEDNUMPOINT_AUTOSCROLL.- autoscale the x-axis for a fixed number of points, with moving maximum and minimum values, use in scrolling graphs.

autoscaleymode

Specifies y-axis auto-scale mode of the scroll frame. Use one of the y-axis scroll frame constants:
RT_NO_AUTOSCALE_Y - no auto-scale for the y-axis
RT_AUTOSCALE_Y_MIN - autoscale y-axis minimum only
RT_AUTOSCALE_Y_MAX - autoscale y-axis maximum only
RT_AUTOSCALE_Y_MINMAX. - - autoscale y-axis minimum and maximum

Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAutoScaleRoundXMode**(value) and value := **getAutoScaleRoundXMode**();

AutoScaleRoundXMode	Get/Set the auto-scale round mode for the x-coordinate. Use one of the AUTOAXES round mode constants: AUTOAXES_EXACT, AUTOAXES_NEAR, AUTOAXES_FAR.
AutoScaleRoundYMode	Get/Set the auto-scale mode for the y-coordinate. Use one of the AUTOAXES round mode constants: AUTOAXES_EXACT, AUTOAXES_NEAR, AUTOAXES_FAR.
EndOfPlotLineMarker .	Get/Set The end of plot marker type. Use one of the Marker marker type constants: MARKER_NULL, MARKER_VLINE, MARKER_HLINE, MARKER_HVLINE.
MaxDisplayHistory	Get/Set the maximum number of points displayed.
MinSamplesForAutoScale	Get/Set the minimum number of samples that need to be in the dataset before an auto-scale operation is carried out. This prevents the first datapoints from generating an arbitrarily small range.
ScrollRescaleMargin	Get/Set the scroll rescale margin. When the limits of the scale

	needs to be increased, the ScrollRescaleMargin * (current range of the x-axis) is added to the upper and lower limits of the current scale.
ScrollScaleModeX	Get/Set the scrolling mode for the x-coordinate. Use one of the x-axis scroll frame constants: RT_NO_AUTOSCALE_X, RT_AUTOSCALE_X_CURRENT_SCALE, RT_AUTOSCALE_X_MIN, RT_AUTOSCALE_X_MAX, RT_AUTOSCALE_X_MINMAX, RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL, RT_MAXEXTENT_FIXEDSTART_AUTOSCROLL, RT_FIXEDNUMPOINT_AUTOSCROLL
ScrollScaleModeY	Get/Set the scrolling mode for the y-coordinate. Use one of the y-axis scroll frame constants: RT_NO_AUTOSCALE_Y, RT_AUTOSCALE_Y_MIN, RT_AUTOSCALE_Y_MAX, RT_AUTOSCALE_Y_MINMAX
TimeStampMode	Get/Set the time stamp mode for the time values in the process variables. Use one of the time stamp mode constants: RT_NOT_MONOTONIC_X_MODE - not monotonic means that the x-values do not have to increase with increasing time. A real-time xy plot that plots x-values against y-values might have this characteristic. RT_MONOTONIC_X_MODE – The default value. Monotonic means that the x-values always increase with increasing time. If the scroll frame routines know that the x-values will never "backtrack" it speeds up the search algorithm for minimum and maximum x-values to use in auto-scaling the x-axis.

A complete listing of **RTScrollFrame** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

See the examples for the **RTSimpleSingleValuePlot** and **RTGroupMultiValuePlot for example of the use of the RTScrollFrame class.**

Class RTVerticalScrollFrame

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTMultiValueIndicator
RTVerticalScrollFrame

The **RTVerticalScrollFrame** is basically the same as the original **RTScrollFrame**, except it controls scrolling along the vertical axis. When you use a vertical scroll frame, typically you would have the y-scale setup as an elapsed time, or time/date based scale. It

can also be setup as a numeric base scale. Otherwise it works much the same as the **RTScrollFrame**.

.

RTVerticalScrollFrame constructors

```
public RTVerticalScrollFrame(
   ChartView component,
   RTProcessVar processvar,
   PhysicalCoordinates initialscale,
   int scrollymode,
   int autoscalexmode
);
public RTVerticalScrollFrame (
   ChartView component,
   RTProcessVar processvar,
   PhysicalCoordinates initialscale,
   int scrollmode
);

public RTVerticalScrollFrame (
   ChartView component,
   PhysicalCoordinates initialscale,
   int scrollymode,
   int autoscalexmode
);
```

Parameters

component

This **ChartView** component the scroll frame is placed in.

processvar

The source process variable.

initialscale

A coordinate system that serves as the initial scale for the scroll frame.

scrollymode

Specifies y-axis auto-scale mode of the scroll frame. Use one of the x-axis scroll frame constants:
RT_NO_AUTOSCALE_X - no auto-scale for the x-axis, use in non-scrolling graphs.
RT_AUTOSCALE_Y_CURRENT_SCALE - auto-scale based on current scale, use in non-scrolling graphs.
RT_AUTOSCALE_Y_MIN - autoscale x-axis minimum only, use in non-scrolling graphs.
RT_AUTOSCALE_Y_MAX - autoscale x-axis maximum only, use in non-scrolling graphs.
RT_AUTOSCALE_Y_MINMAX - autoscale x-axis minimum and maximum, use in non-scrolling graphs.
RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL - autoscale the x-axis for a fixed range, with moving maximum and minimum values, use in scrolling graphs.

RT_MAXEXTENT_FIXEDSTART_AUTOSCROLL - autoscale the x-axis with the start of the x-axis fixed, and the end of the x-axis moving, use in scrolling graphs.
RT_FIXEDNUMPOINT_AUTOSCROLL.- autoscale the x-axis for a fixed number of points, with moving maximum and minimum values, use in scrolling graphs.

autoscalexmode

Specifies x-axis auto-scale mode of the scroll frame. Use one of the y-axis scroll frame constants constants:
RT_NO_AUTOSCALE_X - no auto-scale for the y-axis
RT_AUTOSCALE_X_MIN - autoscale y-axis minimum only
RT_AUTOSCALE_X_MAX - autoscale y-axis maximum only
RT_AUTOSCALE_X_MINMAX. - - autoscale y-axis minimum and maximum

Selected Public Instance Properties

AutoScaleRoundXMode	Get/Set the auto-scale round mode for the x-coordinate. Use one of the AUTOAXES round mode constants: AUTOAXES_EXACT, AUTOAXES_NEAR, AUTOAXES_FAR.
AutoScaleRoundYMode	Get/Set the auto-scale mode for the y-coordinate. Use one of the AUTOAXES round mode constants: AUTOAXES_EXACT, AUTOAXES_NEAR, AUTOAXES_FAR.
MaxDisplayHistory	Get/Set the maximum number of points displayed.
MinSamplesForAutoScale	Get/Set the minimum number of samples that need to be in the dataset before an auto-scale operation is carried out. This prevents the first datapoints from generating an arbitrarily small range.
ScrollRescaleMargin	Get/Set the scroll rescale margin. When the limits of the scale needs to be increased, the ScrollRescaleMargin * (current range of the x-axis) is added to the upper and lower limits of the current scale.
ScrollScaleModeX	Get/Set the scrolling mode for the x-coordinate. Use one of the x-axis scroll frame constants: RT_NO_AUTOSCALE_X, RT_AUTOSCALE_X_MIN, RT_AUTOSCALE_X_MAX, RT_AUTOSCALE_X_MINMAX
ScrollScaleModeY	Get/Set the scrolling mode for the y-coordinate. Use one of the y-axis scroll frame constants: RT_NO_AUTOSCALE_Y, RT_AUTOSCALE_Y_CURRENT_SCALE, RT_AUTOSCALE_Y_MIN, RT_AUTOSCALE_Y_MAX, RT_AUTOSCALE_Y_MINMAX, RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL, RT_MAXEXTENT_FIXEDSTART_AUTOSCROLL, RT_FIXEDNUMPOINT_AUTOSCROLL

TimeStampMode	Get/Set the time stamp mode for the time values in the process variables. Use one of the time stamp mode constants: RT_NOT_MONOTONIC_Y_MODE - not monotonic means that the x-values do not have to increase with increasing time. A real-time xy plot that plots x-values against y-values might have this characteristic. RT_MONOTONIC_Y_MODE – The default value. Monotonic means that the x-values always increase with increasing time. If the scroll frame routines know that the x-values will never "backtrack" it speeds up the search algorithm for minimum and maximum x-values to use in auto-scaling the x-axis.

A complete listing of **RTVerticalScrollFrame** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

See the example VerticalScrolling **for example of the use of the RTVerticalScrollFrame class.**

```
scrollFrame =  new RTVerticalScrollFrame(this, currentTemperature1, pTransform1,
     ChartObj.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL)
scrollFrame.addProcessVar(currentTemperature2)

scrollFrame.setScrollScaleModeX(ChartObj.RT_AUTOSCALE_X_MINMAX);
' Allow 400 samples to accumlate before autoscaling y-axis. This prevents rapid
' changes of the y-scale for the first few samples
scrollFrame.setMinSamplesForAutoScale(400);
scrollFrame.setScrollRescaleMargin(0.05);

chartVu.AddChartObject(scrollFrame);
```

Single Channel Scrolling Graphs

Class RTSimpleSingleValuePlot

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTSingleValueIndicator
RTSimpleSingleValuePlot

The **RTSimpleSingleValuePlot** plot class uses a template based on the **QCChart2D SimplePlot** class to create a real-time plot that displays **RTProcessVar** current and historical real-time data in a scrolling line, scrolling bar, or scrolling scatter plot format. Any plot object derived from the **QCChart2D SimplePlot** can be plotted as a scrolling graph.

RTSimpleSingleValuePlot constructors

```
public RTSimpleSingleValuePlot(
   PhysicalCoordinates transform,
```

```
   SimplePlot plottemplate,
   RTProcessVar datasource
);

public RTSimpleSingleValuePlot(
   SimplePlot plottemplate,
   RTProcessVar datasource
);
```

Parameters

transform

The coordinate system for the new **RTSimpleSingleValuePlot** object.

plottemplate

This **SimplePlot** object is used as a template for the scrolling plot object.

datasource

The source process variable.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setEndOfPlotLineMarker**(value) and value := **getEndOfPlotLineMarker**();

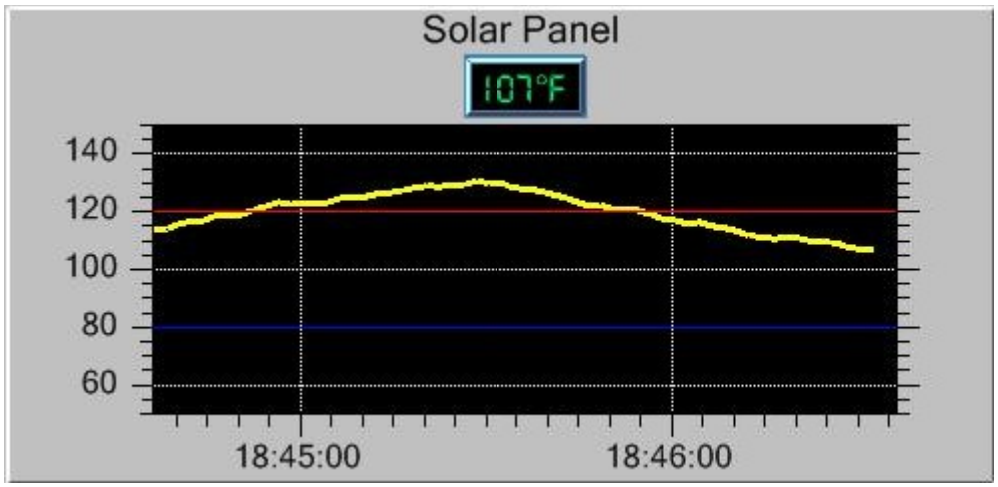
EndOfPlotLineMarker	Get/Set The end of plot marker type. Use one of the Marker marker type constants: MARKER_NULL, MARKER_VLINE, MARKER_HLINE, MARKER_HVLINE.
PlotTemplate	Get/Set the simple plot template.

A complete listing of **RTSimpleSingleValuePlot** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for a simple single-channel scrolling line plot

The example below, extracted from the HomeAutomation example, file SolarPanelControl, method **InitializeScrollGraph**, creates a single-channel scrolling graph.

Note: Both the **RTScrollFrame** and the **RTSimpleSingleValuePlot** objects are added to the **ChartView**. When the **ChartView.updateDraw** method is called, the **RTScrollFrame** object in the **ChartView** object list causes the scroll graph coordinate system to be re-scaled to reflect the current data values. The **RTSimpleSingleValue** object in the **ChartView** list redraws the line plot in the new re-scaled coordinate system.

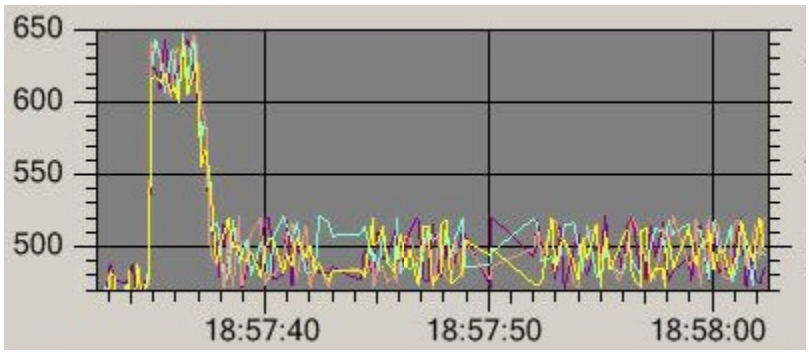



```
scrollFrame = new RTScrollFrame(this, currentTemperature1, pTransform1,
    ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);
scrollFrame.setScrollScaleModeY ( ChartConstants.RT_NO_AUTOSCALE_Y);
scrollFrame.setScrollRescaleMargin ( 0.05);
chartVu.addChartObject(scrollFrame);

SimpleLinePlot lineplot =  new SimpleLinePlot(pTransform1, null, attrib1);
lineplot.setFastClipMode( ChartConstants.FASTCLIP_X);
RTSimpleSingleValuePlot solarPanelLinePlot =
    new RTSimpleSingleValuePlot(pTransform1,lineplot, currentTemperature1);
chartVu.addChartObject(solarPanelLinePlot);
```

Example for a multi-channel scrolling line plot

The example below, extracted from the Dynamometer example, file Dynamometer, method **InitializeEngine1ScrollGraph**, creates a multi-channel scrolling graph.

Note: You do not have to use an **RTGroupMultiValuePlot** to plot multi-channel data in a scrolling graph. You can just use multiple **RTSimpleSingleValuePlot** objects as in the example below. You can also mix object types, including line plots, bar plots and scatter plot in the same scrolling graph.

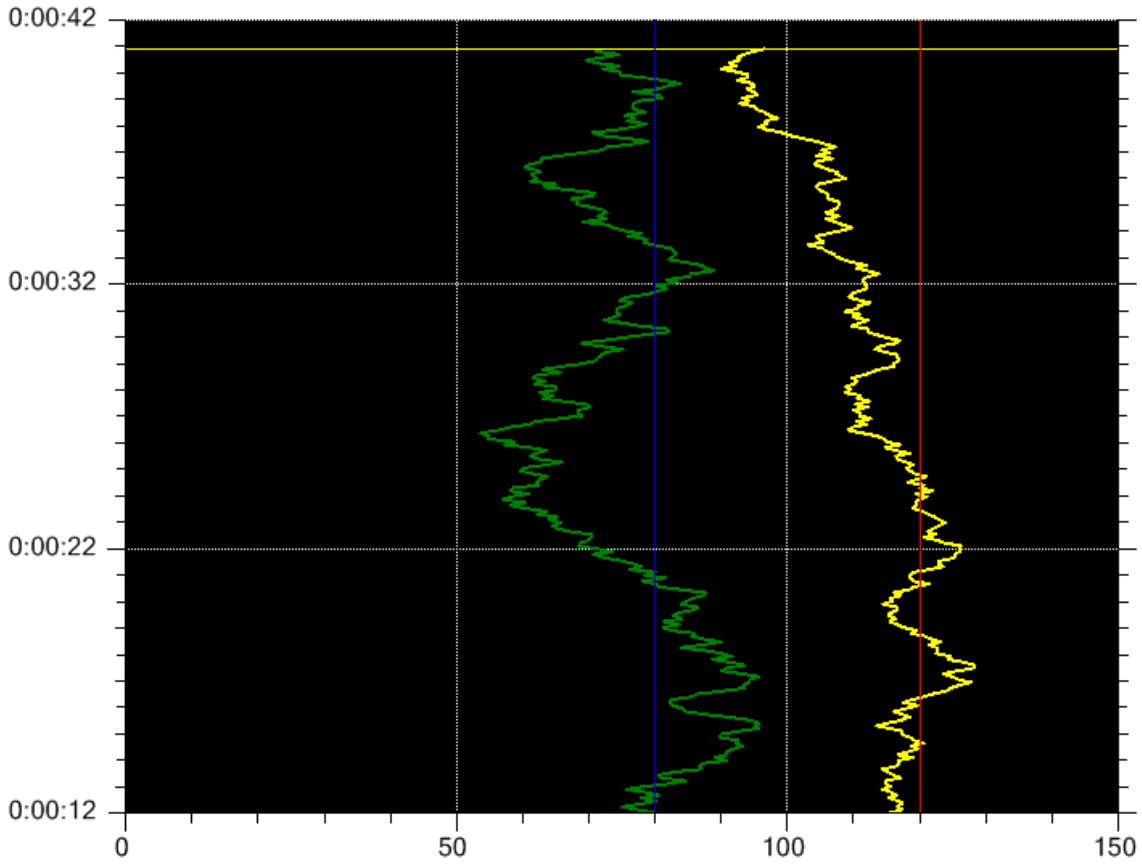


```
scrollFrame1 = new RTScrollFrame(this, EngineCylinderTemp1[0], pTransform1,
ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);

scrollFrame1.addProcessVar(EngineCylinderTemp1[1]);
scrollFrame1.addProcessVar(EngineCylinderTemp1[2]);
scrollFrame1.addProcessVar(EngineCylinderTemp1[3]);

scrollFrame1.setScrollScaleModeY( ChartConstants.RT_AUTOSCALE_Y_MINMAX);
scrollFrame1.setScrollRescaleMargin( 0.05);

chartVu.addChartObject(scrollFrame1);
for (int i=0; i < 4; i++)
{
    SimpleLinePlot lineplot =  new SimpleLinePlot(pTransform1, null,
        attribarray[i]);
    lineplot.setFastClipMode( ChartConstants.FASTCLIP_X);
    rtLinePlotArray1[i] = new RTSimpleSingleValuePlot(pTransform1,lineplot,
    EngineCylinderTemp1[i]);
    chartVu.addChartObject(rtLinePlotArray1[i]);
}
```

Example for a simple two-channel vertical scrolling line plot

The example below, extracted from the VerticalScrolling.ElapsedTimeVerticalScrolling, method **InitializeVerticalScrollGraph**, creates a vertical, elapsed time based, two-channel scrolling graph.

Note: Both the **RTVertcialScrollFrame** and the **RTSimpleSingleValuePlot** objects are added to the **ChartView**. When the **ChartView.UpdateDraw** method is called, the **RTScrollFrame** object in the **ChartView** object list causes the scroll graph coordinate system to be re-scaled to reflect the current data values. The **RTSimpleSingleValue** object in the **ChartView** list redraws the line plot in the new re-scaled coordinate system.




```
scrollFrame = new RTVerticalScrollFrame(this, currentTemperature1, pTransform1,
ChartObj.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);
scrollFrame.addProcessVar(currentTemperature2);

scrollFrame.setScrollScaleModeX (ChartObj.RT_AUTOSCALE_X_MINMAX);
// Allow 100 samples to accumlate before autoscaling y-axis. This prevents rapid
// changes of the y-scale for the first few samples
scrollFrame.setMinSamplesForAutoScale(100);
scrollFrame.setScrollRescaleMargin(0.05);
chartVu.addChartObject(scrollFrame);

ChartAttribute attrib1 = new ChartAttribute(Color.YELLOW, 1,
ChartConstants.LS_SOLID);
SimpleLinePlot lineplot1 = new SimpleLinePlot(pTransform1, null, attrib1);
lineplot1.setCoordinateSwap(true);
RTSimpleSingleValuePlot solarPanelLinePlot1 = new
RTSimpleSingleValuePlot(pTransform1, lineplot1, currentTemperature1);
chartVu.addChartObject(solarPanelLinePlot1);

ChartAttribute attrib2 = new ChartAttribute(Color.GREEN, 1,
ChartConstants.LS_SOLID);
SimpleLinePlot lineplot2 = new SimpleLinePlot(pTransform1, null, attrib2);
lineplot2.setCoordinateSwap(true);
RTSimpleSingleValuePlot solarPanelLinePlot2 = new
RTSimpleSingleValuePlot(pTransform1, lineplot2, currentTemperature2);
chartVu.addChartObject(solarPanelLinePlot2);
```

13. Multi-Channel Scrolling Plots

RTGroupMultiValuePlot

The **RTGroupMultiValuePlot** class can be used to plot multi-channel scrolling plot data. It uses the **QCChart2D GroupPlot** class as a template to define the attributes of the multi-channel plot. It is not the only technique, since the previous chapter had an example that plotted multiple line plots using the **RTSimpleSingleValue** plot class. If you need to plot multiple channel data, and each channel is a different plot type (i.e. one channel is a line plot, the next channel is a bar plot and the third channel is a scatter plot), you must use the technique that uses **RTSimpleSingleValue** objects**.**

There are two basic types of **QCChart2D GroupPlot** objects. The first type is a multi-channel plot. Plot objects of this type include **QCChart2D MultiLinePlot**, **QCChart2D StackedLinePlot**, **QCChart2D GroupBarPlot**, **QCChart2D StackedBarPlot**. These objects are characterized as having unique **ChartAttribute** objects defining the colors and fill characteristic of each channel. The second type is the multi-variable plot. These are objects that require two or more y-values to characterize the plot at a given instance in time. These include the **QCChart2D HistogramPlot, QCChart2D BubblePlot, QCChart2D FloatingBarPlot, QCChart2D CellPlot** and **QCChart2D OHLCPlot** classes**.** Usually one instance of one of these multi-variable objects is characterized by a single **ChartAttribute**, similar to the **QChart2D.SimplePlot** objects. Both types of **QCChart2D GroupPlot** objects can be used in scrolling graphs.

Multi-Channel Scrolling Graphs

Class RTGroupMultiValuePlot

Com.quinncurtis.chart2djava.ChartPlot
RTPlot
RTMultiValueIndicator
RTGroupMultiValuePlot

The **RTGroupMultiValuePlot** plot class uses a template based on the **QCChart2D GroupPlot** class to create a real-time plot that displays a collection of **RTProcessVar** objects as a group plot in a scrolling graph format.

RTGroupMultiValuePlot constructors

```
public RTGroupMultiValuePlot(
   PhysicalCoordinates transform,
   GroupPlot plottemplate,
   RTProcessVar[] datasource
);
```

Parameters

transform

The coordinate system for the new **RTGroupMultiPlot** object.

plottemplate

A template defining the group plot object.

datasource

An array of **RTProcessVar** objects, one for each group in the group plot template.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setEndOfPlotLineMarker**(value) and value := **getEndOfPlotLineMarker**();

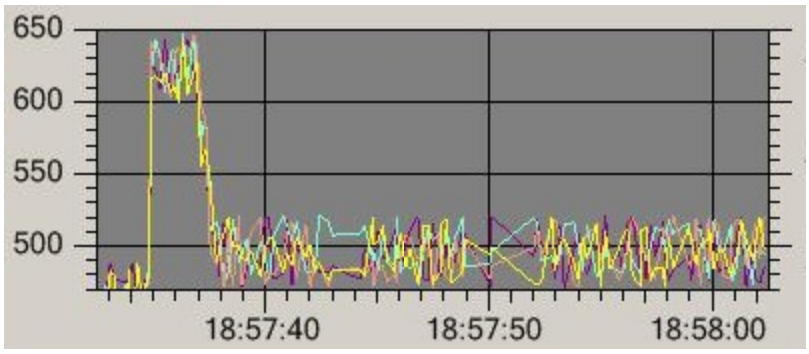
EndOfPlotLineMarker	Get/Set The end of plot marker type. Use one of the Marker marker type constants: MARKER_NULL, MARKER_VLINE, MARKER_HLINE, MARKER_HVLINE.
MarkerGroupNumber	Get/Set the group number that is used for the end of plot line marker.
PlotTemplate	Get/Set the group plot template.

A complete listing of **RTGroupMultiValuePlot** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for a multi-channel scrolling line plot

The example below, extracted from the Dynamometer example, file Dynamometer, method **InitializeEngine2ScrollGraph**, creates a multi-channel scrolling graph.

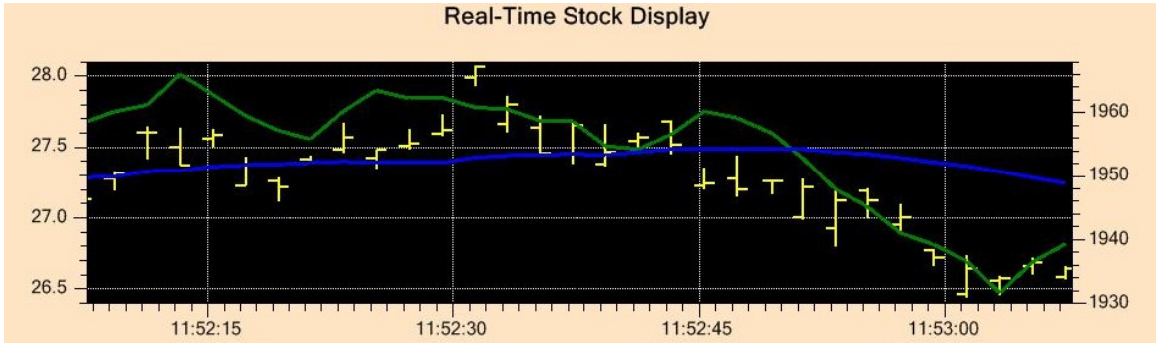
Note: You do not have to use an **RTGroupMultiValuePlot** to plot multi-channel data in a scrolling graph. You can just use multiple **RTSimpleSingleValuePlot** objects as in the InitializeEngine1ScrollGraph method.




```
scrollFrame2 = new RTScrollFrame(this, EngineCylinderTemp2[0],
       pTransform1, ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);

scrollFrame2.addProcessVar(EngineCylinderTemp2[1]);
scrollFrame2.addProcessVar(EngineCylinderTemp2[2]);
scrollFrame2.addProcessVar(EngineCylinderTemp2[3]);

scrollFrame2.setScrollScaleModeY(ChartConstants.RT_AUTOSCALE_Y_MINMAX);
scrollFrame2.setScrollRescaleMargin( 0.05);

// The scrolling graph for Engine #1 was created using for instances of
// SimpleLinePlot. This
// one is created using one GroupLinePlot. They should result in identical
//  displays.
MultiLinePlot multilineplot =  new MultiLinePlot(pTransform1, null, attribarray);
multilineplot.setFastClipMode( ChartConstants.FASTCLIP_X);
rtMultiLinePlot = new RTGroupMultiValuePlot(pTransform1,multilineplot,
     EngineCylinderTemp2);
chartVu.addChartObject(rtMultiLinePlot);
chartVu.addChartObject(scrollFrame2);
```

Example for multi-scale, multi-axis scrolling graph combining stock open-high-low-close plots with line plots.

The example below, extracted from the RTStockDisplay example, method **InitializeScrollGraph**, creates a multi-channel scrolling graph the combines an open-high-low-close plots with line plots using two different scales.




```
scrollFrame1 = new RTScrollFrame(this, stockOpen1, pTransform1,
    ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);
```

```
scrollFrame1.setScrollScaleModeY(ChartConstants.RT_AUTOSCALE_Y_MINMAX);
// Need to add this ProcessVar s to have auto-scale work for all values
//  of OHLC plot
scrollFrame1.addProcessVar(stockHigh1);
scrollFrame1.addProcessVar(stockLow1);
scrollFrame1.addProcessVar(stockClose1);
scrollFrame1.setScrollRescaleMargin(0.05);
chartVu.addChartObject(scrollFrame1);

scrollFrame2 = new RTScrollFrame(this, NASDAQChannel, pTransform2,
       ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);
scrollFrame2.setScrollScaleModeY(ChartConstants.RT_AUTOSCALE_Y_MINMAX);
scrollFrame2.setScrollRescaleMargin(0.05);
chartVu.addChartObject(scrollFrame2);

ChartAttribute attrib1 = new ChartAttribute (Color.yellow,
       2,ChartConstants.LS_SOLID);

OHLCPlot ohlcplot1 = new OHLCPlot(pTransform1, null,
       ChartCalendar.getCalendarWidthValue(ChartConstants.SECOND,0.5),  attrib1);
ohlcplot1.setFastClipMode( ChartConstants.FASTCLIP_X);
RTProcessVar [] stockvars = {stockOpen1, stockHigh1, stockLow1, stockClose1};
rtPlot1 = new RTGroupMultiValuePlot(pTransform1,ohlcplot1, stockvars);
chartVu.addChartObject(rtPlot1);

ChartAttribute attrib2 = new ChartAttribute (Color.green,
       3,ChartConstants.LS_SOLID);

SimpleLinePlot lineplot2 =  new SimpleLinePlot(pTransform2, null, attrib2);
lineplot2.setFastClipMode( ChartConstants.FASTCLIP_X);
rtPlot2 = new RTSimpleSingleValuePlot(pTransform2,lineplot2, NASDAQChannel);
chartVu.addChartObject(rtPlot2);

ChartAttribute attrib3 = new ChartAttribute (Color.blue,
       3,ChartConstants.LS_SOLID);

SimpleLinePlot lineplot3 =  new SimpleLinePlot(pTransform2, null, attrib3);
lineplot3.setFastClipMode( ChartConstants.FASTCLIP_X);
rtPlot3 = new RTSimpleSingleValuePlot(pTransform1,lineplot3, movingAverageStock);
chartVu.addChartObject(rtPlot3);
```

14. Buttons, Track Bars and Other Form Control Classes

RTControlButton
RTControlTrackBar
RTFormControl
RTFormControlPanelMeter
RTFormControlGrid

Real-time displays often require user interface features such as buttons and track bars. The Java Swing package includes a large number of useful controls. The **JButton, JSlider, JScrollBar** are examples of what we refer collectively as Form Controls. Sometime though the Form controls have shortcomings.. The **JScrollBar** and the **JSlider** controls have the fault that they work only with an integer range of values. The **JButton** controls are momentary and require extra programming in order to use them as toggle buttons or radio buttons..

We created subclassed versions of the **JSlider** control and the **JButton** control. Our version of the **JSlider** control is **RTControlTrackBar** and adds floating point scaling for the track bar endpoints, increments, current value and tick mark frequency. Our version of the **JButton** control is **RTControlButton** adds superior On/Off button text and color control and supports momentary, toggle and radio button styles.

Control Buttons

Class RTControlButton

Javax.swing.JButton
RTControlButton

The **RTControlButton** class is subclassed from the Swing **JButton** class. It combines the features of a toggle button and momentary closure button. A toggle button acts more like a check box; when it is pressed it toggles its state to checked or unchecked. A momentary button is more like a regular Java button; when the button is pressed it is only in the checked state while pressed, otherwise it returns to the unchecked state. When an **RTControlButton** is added to an **RTFormControlGrid**, it can also act as a radio button, where all radio buttons in an **RTFormControlGrid** are mutually exclusive. The **RTControlButton** also adds unique color and text properties for the button in both the checked and unchecked state.

RTControlButton constructor

```
public RTControlButton(
```

```
   int buttontype
);
```

Parameters

buttontype

The button type of the new button. User on of the button subtype constants:
RT_CONTROL_RADIOBUTTON_SUBTYPE,
RT_CONTROL_MOMENTARYBUTTON_SUBTYPE,
RT_CONTROL_TOGGLEBUTTON_SUBTYPE.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setButtonChecked**(value) and value := **getButtonChecked**();

ButtonChecked	Get/Set the button check state.
ButtonCheckedColor	Get/Set the color of the button when the button is checked.
ButtonCheckedText	Get/Set the button text when the button is checked.
ButtonCheckedTextColor	Get/Set the color of the button text when the button is checked.
ButtonSubtype	Get/Set the button subtype. RT_CONTROL_RADIOBUTTON_SUBTYPE, RT_CONTROL_MOMENTARYBUTTON_SUBTYPE, RT_CONTROL_TOGGLEBUTTON_SUBTYPE
ButtonUncheckedColor	Get/Set the color of the button when the button is unchecked.
ButtonUncheckedText	Get/Set the button text when the button is unchecked.
ButtonUncheckedTextColor	Get/Set the color of the button text when the button is unchecked.

A complete listing of **RTControlButton** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for momentary and toggle buttons

The example below, extracted from the ProcessMonitoring example creates three **RTControlButton** buttons; two are momentary buttons and one is a toggle button. The buttons are added to an **RTFormControlGrid** in order to position them as a logical group.

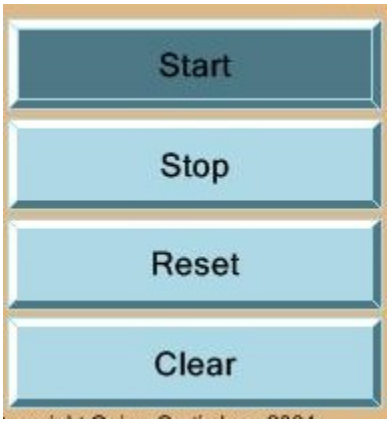

```
RTControlButton ResetErrorTerm = new
    RTControlButton(ChartConstants.RT_CONTROL_MOMENTARYBUTTON_SUBTYPE);
```

```
RTControlButton ResetAll = new
    RTControlButton(ChartConstants.RT_CONTROL_MOMENTARYBUTTON_SUBTYPE);
RTControlButton StartControl = new
    RTControlButton(ChartConstants.RT_CONTROL_TOGGLEBUTTON_SUBTYPE);
.
.
public void InitializeStartStopButtons()

{

        ChartAttribute attrib1 = new ChartAttribute (Color.white, 1,

                ChartConstants.LS_SOLID, Color.red);

        Font buttonfont = font9Bold;

        ChartView chartVu  = this;

        Vector buttonlist = new Vector();

        StartControl.setButtonUncheckedText( "Start");

        StartControl.setButtonCheckedText("Stop");

        StartControl.setButtonFont( buttonfont);

        StartControl.setButtonChecked( false);

        buttonlist.add(StartControl);



        ResetErrorTerm.setButtonUncheckedText( "Reset Error");

        ResetErrorTerm.setButtonFont(buttonfont);

        ResetErrorTerm.setButtonChecked( false);



        buttonlist.add(ResetErrorTerm);

        ResetAll.setButtonUncheckedText( "Reset All");

        ResetAll.setButtonFont( buttonfont);

        ResetAll.setButtonChecked( false);



        buttonlist.add(ResetAll);



        int numColumns = 3;

        int numRows = 1;

        CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);

        pTransform1.setGraphBorderDiagonal(0.72, .555, 0.99, 0.605) ;
```

```
RTFormControlGrid controlgrid =

        new RTFormControlGrid(pTransform1, null, buttonlist, numColumns,

        numRows,   attrib1);

controlgrid.setCellRowMargin( 0.00);

controlgrid.setCellColumnMargin( 0.02);

controlgrid.getFormControlTemplate().setFrame3DEnable( true);

chartVu.addChartObject(controlgrid);
```

Example for momentary and radio buttons

The example below, extracted from the FetalMonitor example, creates four **RTControlButton** buttons; two are momentary buttons and two are radio buttons. The buttons are added to an **RTFormControlGrid** in order to position them as a logical group.




```
RTControlButton StartButton = new
    RTControlButton(ChartConstants.RT_CONTROL_RADIOBUTTON_SUBTYPE);
RTControlButton StopButton = new
    RTControlButton(ChartConstants.RT_CONTROL_RADIOBUTTON_SUBTYPE);
RTControlButton ResetButton = new
    RTControlButton(ChartConstants.RT_CONTROL_MOMENTARYBUTTON_SUBTYPE);
RTControlButton ClearButton = new
    RTControlButton(ChartConstants.RT_CONTROL_MOMENTARYBUTTON_SUBTYPE);
RTFormControlGrid startStopControlGrid;

public void InitializeStartStopButtons()
{  Font buttonfont = font12Bold;

    ChartView chartVu  = this;

    CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
    pTransform1.setGraphBorderDiagonal(0.01, .65, 0.2, 0.98) ;
    ChartAttribute attrib1 = new ChartAttribute (Color.black, 5,
        ChartConstants.LS_SOLID, lightBlue);
```

```
Vector buttonlist1 = new Vector();

StartButton.setButtonUncheckedText("Start");
StartButton.setButtonChecked(true);
StartButton.addMouseListener(this);
StartButton.setButtonFont(buttonfont);

buttonlist1.add(StartButton);

StopButton.setButtonUncheckedText("Stop");
StopButton.setButtonChecked(false);
StopButton.addMouseListener(this);
StopButton.setButtonFont(buttonfont);

buttonlist1.add(StopButton);

ResetButton.setButtonUncheckedText("Reset");
ResetButton.setButtonChecked(false);
ResetButton.addMouseListener(this);
ResetButton.setButtonFont(buttonfont);

buttonlist1.add(ResetButton);

ClearButton.setButtonUncheckedText("Clear");
ClearButton.setButtonChecked(false);
ClearButton.addMouseListener(this);
ClearButton.setButtonFont(buttonfont);

buttonlist1.add(ClearButton);

int numColumns = 1;
int numRows = 4;

startStopControlGrid = newRTFormControlGrid(pTransform1, null, buttonlist1,
       numColumns, numRows,   attrib1);
startStopControlGrid.setCellRowMargin(0.1);
startStopControlGrid.setCellColumnMargin(0.0);
startStopControlGrid.getFormControlTemplate().setFrame3DEnable(flag3DBorder);
chartVu.addChartObject(startStopControlGrid);

Vector buttonlist2 = new Vector();

ChartAttribute attrib2 = new ChartAttribute (Color.black, 5,
    ChartConstants.LS_SOLID, lightGreen);

CartesianCoordinates pTransform2 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
pTransform2.setGraphBorderDiagonal(0.21, .65, 0.4, 0.98) ;

PrimaryLineButton.setButtonUncheckedText("Primary Line");
PrimaryLineButton.setButtonChecked(false);
PrimaryLineButton.addMouseListener(this);
PrimaryLineButton.setButtonFont(buttonfont);

buttonlist2.add(PrimaryLineButton);

SecondaryLineButton.setButtonUncheckedText("Secondary Line");
SecondaryLineButton.setButtonChecked(false);
SecondaryLineButton.addMouseListener(this);
SecondaryLineButton.setButtonFont(buttonfont);

buttonlist2.add(SecondaryLineButton);

Concurrent.setButtonUncheckedText("Concurrent");
Concurrent.setButtonChecked(false);
Concurrent.addMouseListener(this);
Concurrent.setButtonFont(buttonfont);

buttonlist2.add(Concurrent);

numColumns = 1;
```

```
    numRows = 4;

    RTFormControlGrid controlgrid2 = new  RTFormControlGrid(pTransform2, null,
        buttonlist2, numColumns,      numRows,   attrib2);
    controlgrid2.setCellRowMargin(0.1);
    controlgrid2.setCellColumnMargin(0.0);
    controlgrid2.getFormControlTemplate().setFrame3DEnable(flag3DBorder);
    chartVu.addChartObject(controlgrid2);

    }
```

Control TrackBars

Class RTControlTrackBar

Javax.swing.JSlider
RTControlTrackBar

The **RTControlTrackBar** class is subclassed from the Swing **JSlider** class. Our version of the **JSlider** control adds floating point scaling for the track bar endpoints, increments, current value and tick mark frequency.

RTControlButton constructor

```
public RTControlTrackBar(
   double minvalue,
   double maxvalue,
   double largechange,
   double smallchange,
   double tickfrequency
);

public RTControlTrackBar(
   double minvalue,
   double maxvalue
);
```

Parameters

minvalue
Specifies the floating point minimum value for the track bar.
maxvalue
Specifies the floating point maximum value for the track bar.
largechange
Specifies the floating point value for the slider major tick marks
smallchange
Specifies the floating point value for the slider minor tick marks
tickfrequency
Unused.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setRTOrientation**(value) and value := **getRTOrientation**();

RTOrientation	Get/Set a value indicating the horizontal or vertical orientation (**Adjustable.Horizontal** or **Adjustable.Vertical**) of the track bar.
RTLargeChange	Specifies the floating point large change value for the track bar. Equivalent to the **JSlider MajorTickSpacing** property, except allows floating point numbers.
RTMaximum	Specifies the floating point maximum value for the track bar.
RTMinimum	Specifies the floating point minimum value for the track bar.
RTSmallChange	Specifies the floating point small change value for the track bar. . Equivalent to the **JSlider MinorTickSpacing** property, except allows floating point numbers.
RTValue	Specifies the double value of the **RTControlTrackBar** slider.

A complete listing of **RTControlTrackBar** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for single RTControlTrackBar combined with an RTNumericPanelMeter

The example below, extracted from the Treadmill example, creates a single **RTControlTrackBar** and positions a large numeric readout of the trackbar value next to it.


```
public void InitializeLeftPanelMeters()
```

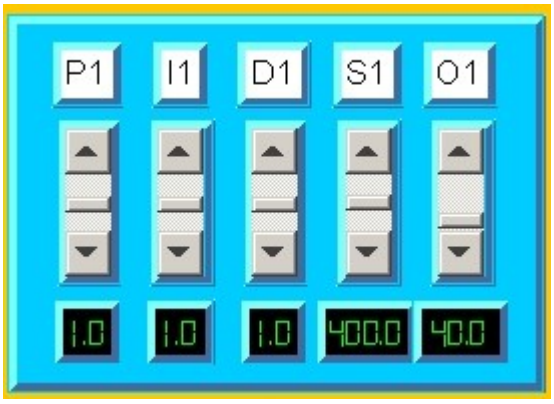
```
{
    Font trackbarfont = font64Numeric;
    Font trackbarTitlefont = font12Bold;

    ChartView chartVu  = this;

    CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);

    pTransform1.setGraphBorderDiagonal(0.01, .12, 0.07, 0.3) ;

    ChartAttribute attrib1 = new ChartAttribute (lightBlue, 3,
        ChartConstants.LS_SOLID, lightBlue);

    runnersPaceTrackbar = new RTControlTrackBar(0.0, 15.0, 5.0, 1.0, 1);
    runnersPaceTrackbar.setOrientation(RTControlTrackBar.VERTICAL);
    runnersPaceTrackbar.setPaintTicks(true);
    runnersPaceTrackbar.setPaintLabels(true);

    runnersPaceTrackbar.setRTValue(3); // MUST USE RTValue to set double value

    RTFormControlPanelMeter formControlTrackBar1 =
        new RTFormControlPanelMeter(pTransform1, runnersPaceTrackbar, attrib1);
    formControlTrackBar1.setRTDataSource(runnersPace);
    formControlTrackBar1.setPanelMeterPosition(ChartConstants.CUSTOM_POSITION);
    formControlTrackBar1.setLocation(0,0.0, ChartConstants.PHYS_POS);
    formControlTrackBar1.setFormControlSize(new ChartDimension(1.0,1.0));
// Must be in same units as SetLocation

    ChartAttribute panelmeterattrib =
        new ChartAttribute(steelBlue,3,ChartConstants.LS_SOLID, Color.black);
    RTNumericPanelMeter panelmeter1 = new RTNumericPanelMeter(pTransform1,
        runnersPace,panelmeterattrib);
    panelmeter1.getNumericTemplate().setTextFont(trackbarfont);
    panelmeter1.getNumericTemplate().setDecimalPos(1);
    panelmeter1.setPanelMeterPosition(ChartConstants.RIGHT_REFERENCED_TEXT);
    panelmeter1.setPositionReference( formControlTrackBar1);
    formControlTrackBar1.addPanelMeter(panelmeter1);

    ChartAttribute panelmetertagattrib = new ChartAttribute(beige,0,
        ChartConstants.LS_SOLID, beige);
    RTStringPanelMeter panelmeter2 = new RTStringPanelMeter(pTransform1, runnersPace,
        panelmetertagattrib, ChartConstants.RT_TAG_STRING);
    panelmeter2.setPositionReference(panelmeter1);
    panelmeter2.getStringTemplate().setTextFont(trackbarTitlefont);
    panelmeter2.setPanelMeterPosition(ChartConstants.BELOW_REFERENCED_TEXT);
    panelmeter2.setTextColor(Color.black);
    formControlTrackBar1.addPanelMeter(panelmeter2);

    chartVu.addChartObject(formControlTrackBar1);
.
.
.

}
```

Example for multiple RTControlScrollbar controls in an RTFormControlGrid

The example below, extracted from the ProcessMonitoring example creates four **RTControlScrollBar** controls. The scrollbars are added to an **RTFormControlGrid** in order to position them as a logical group.

Note: If an **RTNumericPanelMeter** template is applied to a **RTControlTrackBar** or **RTControlScrollBar** controls in an **RTFormControlGtrid**, they will all end up with the same

number of digits to the right of the decimal, since one template applies to all of the track bars. If the dynamic range of the track bars different enough to require unique decimal precision settings, separate them into different grids.



See the method InitializePIDParameterTrackbars() in the ProcessMonitoring example for the source to this example.

Form Control Panel Meter

This panel meter class encapsulates Form Control objects, including our own **RTControlButton** and **RTControlTrackBar** objects in a panel meter class, so that controls can be added to indicator objects.

Class RTFormControlPanelMeter

RTPanelMeter
RTFormControlPanelMeter

```
public RTFormControlPanelMeter(
   PhysicalCoordinates transform,
   RTProcessVar datasource,
   Control formcontrol,
   ChartAttribute attrib
);

public RTFormControlPanelMeter(
   PhysicalCoordinates transform,
   Control formcontrol,
   ChartAttribute attrib
);
```

Parameters

transform

The coordinate system for the new **RTFormControlPanelMeter** object.

datasource

The process variable associated with the control.

formcontrol

A reference to the Control assigned to this panel meter.

attrib

The color attributes of the panel meter.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setChartObjAttributes**(value) and value := **getChartObjAttriburtes**();

ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a **ChartAttribute** object.
ChartObjScale (inherited from **GraphObj**)	Sets the reference to the **PhysicalCoordinates** object that the chart object is placed in
ControlSizeMode	Set/Get to the size mode for the Control. Use one of the Control size mode constants: RT_ORIG_CONTROL_SIZE, RT_MIN_CONTROL_SIZE, RT_INDICATORRECT_CONTROL_SIZE.
FormControlSize	Get the size of the form control in device units.
Frame3DEnable (inherited from **RTPanelMeter**)	Set/Get to true to enable a 3D frame for the panel meter.
IndicatorRect (inherited from **RTPanelMeter**)	Get/Set Indicator positioning rect.
PanelMeterNudge (inherited from **RTPanelMeter**)	Set/Get the xy values of the panelMeterNudge property. It moves the relative position, using window device coordinates, of the text relative to the specified location of the text.
PanelMeterPosition (inherited from **RTPanelMeter**)	Set/Get the panel meter position value. See the panel meter position table in the **RTPanelMeter** chapter.
PanelMeterRectangle (inherited from **RTPanelMeter**)	Set/Get the panel meter bounding rectangle.
PanelMeterTemplate (inherited from **RTPanelMeter**)	Get a **ChartLabel** object representing the panel meters template.
PositionReference (inherited from **RTPanelMeter**)	Set/Get an **RTPanelMeter** object used as a positioning reference for this RTPanelMeter object.
PositionType (inherited from **GraphObj**)	Get/Sets the current position type.
PrimaryChannel (inherited from	Set/Get the primary channel of the indicator.

RTPlot)	
RTDataSource (inherited from **RTSingleValueIndicator**)	Get/Set the array list holding the **RTProcessVar** variables for the indicator.

A complete listing of **RTFormControlPanelMeter** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for RTControlPanelMeter encapsulation an RTControlTrackBar

The example below, extracted from the Polygraph example, creates an **RTControlPanelMeter** using an **RTControlTrackBar**.

```
RTControlTrackBar timeAxisControlTrackbar;
.
.
public void InitializeTimeAxisTrackbar()
{
    ChartAttribute attrib1 = new ChartAttribute (Color.white, 2,
        ChartConstants.LS_SOLID, coral);

    ChartView chartVu  = this;

    CartesianCoordinates pTransform2 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
    pTransform2.setGraphBorderDiagonal(0.7, 0.90, 0.98, 0.96) ;

    ChartAttribute tbattrib = new ChartAttribute (lightBlue, 2,
        ChartConstants.LS_SOLID, lightBlue);

    double starttime = ChartCalendar.getCalendarMsecs(origStartTime);
    double endtime = ChartCalendar.getCalendarMsecs(origEndTime);
    double range = endtime - starttime;
    timeAxisControlTrackbar = new RTControlScrollBar(0, 100, 10, 1, 10);
    timeAxisControlTrackbar.setOrientation(Adjustable.HORIZONTAL);
    timeAxisControlTrackbar.setRTValue(100); // MUST USE RTValue to set double value
    timeAxisControlTrackbar.addAdjustmentListener(this);

    RTFormControlPanelMeter timeAxisControlPanelTrackBar =
        new RTFormControlPanelMeter(pTransform2, timeAxisControlTrackbar, tbattrib);

    timeAxisControlPanelTrackBar.setPanelMeterPosition(ChartConstants.CUSTOM_POSITION);
    timeAxisControlPanelTrackBar.setLocation(0,0.0);
    timeAxisControlPanelTrackBar.setFormControlSize(new ChartDimension(1.0,1.0));
    chartVu.addChartObject(timeAxisControlPanelTrackBar);
}
```

Form Control Grid

The **RTFormControlGrid** organizes a collection of **Form Control** objects functionally and visually in a grid format. An **RTControlButton** must be added to an **RTFormControlGrid** before the radio button processes of the **RTControlButton** will work.

Class RTFormControlGrid

RTMultiValueIndicator
    RTFormControlGrid

RTFormControlGrid constructors

```
public RTFormControlGrid(
   PhysicalCoordinates transform,
   RTProcessVar[] datasource,
   Vector formcontrolarray,
   int numcols,
   int numrows,
   string[] colheads,
   string[] rowheads,
   ChartAttribute attrib
);

public RTFormControlGrid(
   PhysicalCoordinates transform,
   RTProcessVar[] datasource,
   Vector formcontrolarray,
   int numcols,
   int numrows,
   ChartAttribute attrib
);

public RTFormControlGrid(
   PhysicalCoordinates transform,
   RTProcessVar[] datasource,
   Vector formcontrolarray,
   ChartAttribute attrib
);
```

Parameters

transform
    The coordinate system for the new **RTFormControlGrid** object.
datasource
    An array of the process variables associated with the control grid objects.
formcontrolarray
    An array of the Controls assigned to the control grid.
numcols
    The number of columns in the control grid.
numrows
    The number of rows in the control grid.
colheads
    An array of string that is used as the column heads for the control grid.

rowheads

An array of string that is used as the row heads for the control grid.

attrib

A single attribute object that applies to all control grid objects.

Selected Public Instance Properties*

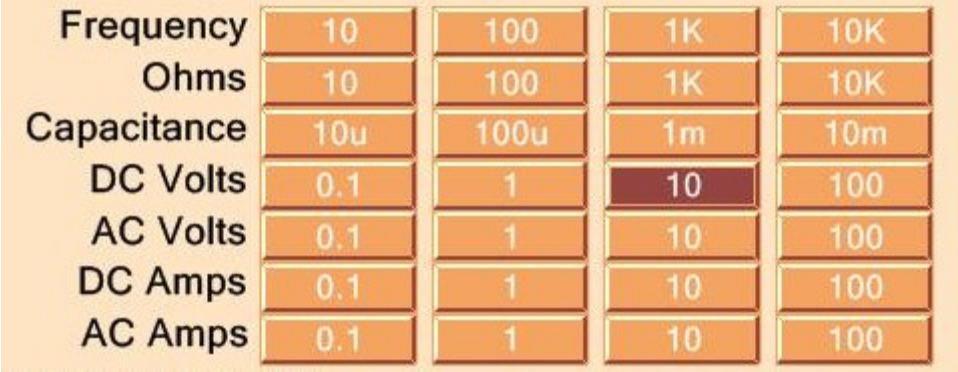
* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setCellColumnMargin**(value) and value := **getCellColumnMargin**();

CellColumnMargin	Get/Set the extra space between columns of the grid, specified in normalized NORM_PLOT_POS coordinates.
CellRowMargin	Get/Set the extra space between rows of the grid, specified in normalized NORM_PLOT_POS coordinates.
ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a ChartAttribute object.
ChartObjScale (inherited from **GraphObj**)	Sets the reference to the **PhysicalCoordinates** object that the chart object is placed in
ColumnHeaders	Set/Get the column headers.
FormControlTemplate	Set/Get the row headers.
HeadersTemplate	Set/Get the string template.
InternalAction	Set/Get to true if you want radio button click and scroll bar value changed events processed to update colors of buttons and numeric values of scroll bars.
NumberColumns	Get the number of rows in the annunciator.
NumberRows	Get the number of rows in the annunciator.
NumChannels (inherited from **RTPlot**)	Get the number of channels in the indicator.
PanelMeterList (inherited from **RTPlot**)	Set/Get the panel meter list of the RT Plot.
PositionType (inherited from **GraphObj**)	Get/Sets the current position type.
RadioButtonChecked	Get/Set the extra space between columns of the grid, specified in normalized NORM_PLOT_POS coordinates.
RowHeaders	Set/Get the row headers.
RTDataSource (inherited from **RTMultiValueIndicator**)	Get/Set the array list holding the **RTProcessVar** variables for the indicator.

A complete listing of **RTFormControlGrid** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for RTFormControlGrid encapsulation RTControlButton objects.

The example below, extracted from the MiniScope example, creates an **RTFormControlGrid** using a collection of **RTControlButtons**.




```
    Vector rangeSelectorButtons = new Vector();
.
.
.
public void InitializeRangeSelectorButtons()
{
        String [] selectorStrings = {"10", "100", "1K", "10K",
                                     "10", "100", "1K", "10K",
                                     "10u", "100u", "1m", "10m",
                                     "0.1","1","10", "100",
                                     "0.1","1","10", "100",
                                     "0.1", "1","10", "100",
                                     "0.1", "1","10", "100"};
    String [] rowStrings = {"Frequency","Ohms", "Capacitance", "DC Volts", "AC Volts",
        "DC Amps", "AC Amps"};
    String[] colStrings = {"","","",""};
    RTControlButton rtbutton;
    ChartView chartVu  = this;
    Font buttonfont = font12Bold;

    CartesianCoordinates pTransform1 = new CartesianCoordinates( 0.0, 0.0, 1.0, 1.0);
    pTransform1.setGraphBorderDiagonal(0.25, .68, 0.95, 0.97) ;
    ChartAttribute attrib1 = new ChartAttribute (Color.white, 3,
        ChartConstants.LS_SOLID, sandyBrown);

    for (int i=0; i < selectorStrings.length; i++)
    {
        rtbutton = new RTControlButton(ChartConstants.RT_CONTROL_RADIOBUTTON_SUBTYPE);
        rtbutton.setButtonUncheckedText(selectorStrings[i]);
        if (i == currentRangeSelectorIndex)
            rtbutton.setButtonChecked( true);
        else
            rtbutton.setButtonChecked( false);
        rtbutton.addMouseListener(this);
        rtbutton.setButtonFont(buttonfont);
        rangeSelectorButtons.add(rtbutton);
    }
```

```
    int numColumns = 4;
    int numRows = 7;

    RTFormControlGrid controlgrid = new  RTFormControlGrid(pTransform1, null,
        rangeSelectorButtons, numColumns,
        numRows,  colStrings, rowStrings, attrib1);
    controlgrid.setCellRowMargin(0.05);
    controlgrid.setCellColumnMargin(0.1);
    controlgrid.getFormControlTemplate().setFrame3DEnable( true);
    controlgrid.getHeadersTemplate().setLineColor(Color.black);
    controlgrid.getHeadersTemplate().setTextFont(font14Bold);
    chartVu.addChartObject(controlgrid);
}
```

15. PID Control

Theory

Proportional-Integral-Derivative (PID) control algorithm is used to drive the process variable (measurement) to the preset value (setpoint).

Temperature control is the most common form of closed loop control. For example, in a simple temperature control system the temperature of a vat of material is to be maintained at a given setpoint, s(t). The output of the controller sets the valve of the actuator to apply less heat to the vat if the current temperature of the vat is greater than the setpoint and more heat to the vat if the current temperature is less than the setpoint.

The PID algorithm calculates its output by summing three terms. One term is proportional to the error (error is defined as the setpoint minus the current measured value). The second term is proportional to the integral of the error over time, and the third term is proportional to the rate of change (first derivative) of the error. The general form of the PID control equation in analog form is:

Eqn. 1

$$m(t) = K_c * \left(e(t) + K_i \int e(t)dt + K_d \frac{de}{dt} \right)$$

proportional *integral* *derivative*

where:
m(t) = controller output deviation
K_c = proportional gain
K_i = reset multiplier (integral time constant)
K_d = derivative time constant
S(t) = current process setpoint
X(t) = actual process measured variable (temperature, for example)
e(t) = error as a function of time = S(t) - X(t)

The variables K_c, K_i and K_d are adjustable and are used to customize a controller for a given process control application. The K_i constant term is listed in some textbooks as $(1/K_i)$. It is simply a matter of the units K_i is specified in. In the K_i form, the units are repeats per minute while in the $1/K_i$ form the units are minutes per repeat (also called reciprocal time). The K_i version presented here is preferred because increasing K_i will increase the integral gain action, just like increasing K_c and K_d will increase the

proportional gain and derivative gain action. If $1/K_i$ is used, then decreasing values of K_i will increase the amount of integral gain.

The proportional term of the PID equation contributes an amount to the controller output directly proportional to the current process error. For example, if the setpoint of the process is 100 degrees and the current temperature of the process is 90 degrees, then the current process error is 10 degrees. The proportional term adds to the controller output an amount equal to $K_c * 10$. The gain term K_c determines how much the control output should change in response to a given error level. If the error is 10, a K_c gain of 0.5 will add 5 to the output of the controller, while a gain of 3 will add 30 to the output of the controller. The larger the value of K_c, the harder the system reacts to differences between the setpoint and the actual temperature. A PID controller can be used as a simple proportional controller by setting the reset rate and derivative time values to 0.0.

Simple proportional control cannot take into account load changes in the process under control. An example of a load for a temperature control loop is the ambient temperature of the room the process is in. The lower the ambient temperature of the room, the larger is the heat loss in the room. It will take more energy to maintain the vat at a given temperature in a cold room than in a warm room. A simple proportional controller cannot account for load changes which take place while the system is under control. Integral control converts the first-order proportional controller into a second order system which is capable of tracking process disturbances. It adds to the controller output a term that corrects for any changes in the load level of the system under control. This integral term is proportional to the sum of all previous process errors in the system. As long as there is a process error, the integral term will add more and more to the controller output until the sum of all previous errors is zero.

The term 'reset rate' is used to describe the integral action of a PID controller. The term derives from the output response of a PI controller (the derivative term set to zero in this case) to a step change in the process error signal. The response consists of an initial jump from zero to the proportional gain setting K_c, followed by a ramp (the integrating action of the integral term) which adds the initial proportional response each integral time T. Therefore the reset rate is defined as the repeats per minute of the initial proportional response.

For example, if the reset rate is 1.0, then for every minute that the error signal is non-zero, the amount of corrective action added to the controller output by the integral term will be equal to the amount added by the proportional term alone. The higher the reset rate, the harder the system will react to non zero error terms.

The addition of the derivative term to the PI controller described above results in the classic three mode PID controller. The derivative action of a PID controller adds to the controller output the value proportional to the slope (rate of change) of the process error. The derivative term "anticipates" the error, providing a harder control response when the error term is going in the wrong direction and a dampening response when the error term is going in the right direction. It reduces overshoot for transient upsets. Proper use of the derivative term can result in much faster process response.

Computer based versions of the PID algorithm are based on sampled data discrete time control theory. The discrete time equivalent of the PID equation is:

Eqn. 2

$$m(i) = K_c * (e(i) + T * K_i \sum_{k=0}^{i} e(k) + (k_d/T) * (e(i)-e(i-1))$$

where
T = sampling interval
e(i) = error at ith sampling interval = S(t) -X(t)
e(i-1) = error at previous sampling interval
m(i) = controller output deviation
K_c = proportional gain
K_i = integral action time
K_d = derivative action time

The proportional term is the same between the Eqn. 1 and Eqn. 2. The integral term of the first equation is replaced by a summation term and the derivative term is replaced by the a first order difference approximation. In actual practice, the first order difference term, $(e_i - e_{i-1})$, is very susceptible to noise problems. In most practical systems this term is replaced by the more stable, higher order equation:

$$\Delta e = (e(i) + 3 * e(i-1) – 3 * e(i-2) – e(i-3))/6$$

A common problem in discrete control systems arises from the summation of the error term for the integral action of the control equation. If a process maintains an error for a long period of time, it is possible that this summation can build to a very large numerical value. Even though the error term returns to zero or moves in the opposite direction, it will take a very long time to reduce the sum below the D/A saturation levels. Practical systems stop the summation of error terms if the current PID output level is outside a user specified range of high and low output values. This limiting of the summation term is commonly referred to as anti-reset-windup.

Implementation

Real-Time Graphics Tools for Windows can maintain an unlimited number of control loops simultaneously; the only limit being memory and CPU power. A PID control object (the terms *PID controller* and *PID object* are used interchangeably in this documentation) is created and configured using the RTPID class. The **rTCalcPID** function calculates the

PID algorithm's output. It should be called at equal time intervals. PID algorithm constants can be tuned by adjusting corresponding property values.

A typical problem occurs when a PID object is switched from manual to automatic mode or when a PID constant is changed: the output value can change very quickly, possibly damaging the control equipment. The Quinn-Curtis implementation of the PID algorithm uses the "bumpless transfer" technique to prevent this problem. The algorithm also uses *anti-reset-windup* technique.

PID Control

Class RTPIDControl

RTPIDControl constructors

```
public RTPIDControl(
   int ptype,
   double setpnt,
   double steadstat,
   double prop,
   double integ,
   double deriv,
   double lowclmp,
   double highclmp,
   double rateclmp,
   double sampleper,
   double filterconst
);

public RTPIDControl(
   double setpnt,
   double steadstat,
   double prop,
   double integ,
   double deriv,
   double sampleper,
   double filterconst
);
```

Parameters

setpnt

Specifies the desired value for the process variable.

steadstat

Anticipated steady state value for the output, also known as bias. If you do not know the steady state value, use 0.0 for this parameter. Setting this value properly improves response because it does not have to rely on integral response, starting with a zero initial error summation term, to add enough to the control output to make up for system losses.

prop

Specifies the proportional gain constant. The proportional term adjust the output value proportional to the current error term.

integ

Specifies the integral gain constant. The integral term adjusts the output value by accumulating, or integrated the error term over time.

deriv

Specifies the derivative gain constant. The derivative term looks at the rate of change of the input and adjusts the output based on the rate of change. The derivative function uses the time derivative of the error term.

lowclmp

Specifies the low clamping value for output. If the output of the PID calculation results in a value less than *lowclmp*, the value will be clamped to *lowclmp*

highclmp

Specifies the high clamping value for output. If the output of the PID calculation results in a value higher than *highclmp*, the value will be clamped to *highclmp.*

rateclmp

Clamping limit for the output rate of change, measured in output units per minute. It limits the rate of change of the algorithm output.

sampleper

Sample period of PID updates, in minutes. For example, if the controller's output is calculated two times a second, the value of this parameter is 1 / (2 * 60) = 0.0084 minutes

filterconst

A value in the range 0.0 to 1.0, affecting the filtering of the noisy measurement signal. A value of 0.0 means that no filtering takes place. The filtering effect is maximal when *rFiltConst* is 1.0. The formula for filtering is:

Filtered value = (1.0 - *rFiltConst)* * Measured value + *rFiltConst* * (Previous filtered value)

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setDerivativeConstant**(value) and value := **getDerivativeConstant**();

DerivativeConstant	Set/Get derivative constant value
E1	Set/Get error term t-1.
E2	Set/Get error term t-2.

E3	Set/Get error term t-3.
HighClamp	Set/Get high clamping value for output .
IntegralConstant	Set/Get integral constant value
LastMv	Set/Get previous exponential smoothing constant.
LastPIDValue	Set/Get previous PID output value.
LowClamp	Set/Get low clamping value for output .
MvFilter	Set/Get exponential smoothing constant.
NewError	Set/Get new error value.
OldError	Set/Get previous error value.
ProportionalConstant	Set/Get proportional constant value
RateClamp	Set/Get rate (first derivative of output) clamping value for output .
SamplePeriod	Set/Get time between adjacent updates.
SetPoint	Set/Get setpoint value.
SteadyState	Set/Get steady state output position.
SumError	Set/Get sum of all previous errors .

Selected Public Instance Methods

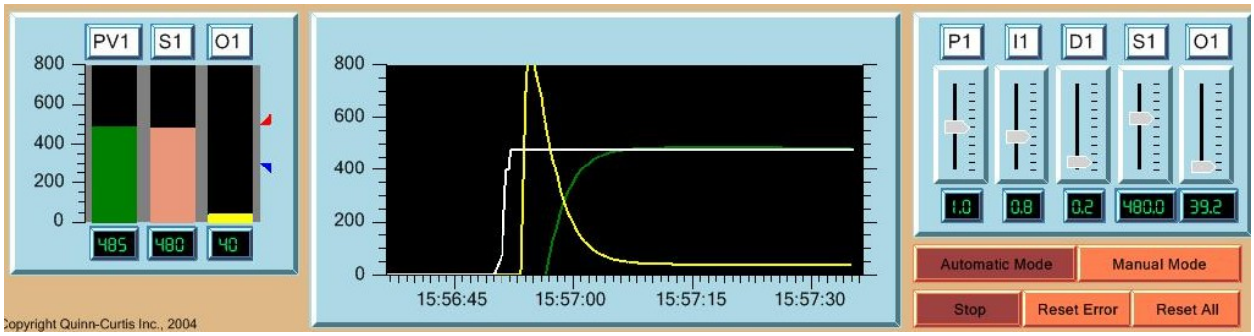
rTCalcPID	This method performs a PID loop calculation.
rTResetErrorTerms	This method resets all of the error terms for the PID calculations.
updatePIDIntermediateParameters	This method updates the intermediate values in the PID calculation.

A complete listing of **RTPIDControl** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

The output of the PID equation, calculated using the **RTCalcPID** method, is in the same units as the measured variable. If the measured variable is temperature with a potential range of 0-300, then the output of the PID equation will also be temperature in the same range, though it can have even wider swings than the measured variable. The output of the PID equation is expected to drive some sort of control device, either an actuator, heater, pressure control valve, or dc servomotor, which has completely different units than the control output. It is up to the control engineer to calculate the transfer function, usually a basic mx + b equation, to the control output so that it matches the input range of the control device, whether it be 0-10V, 4-20mA or some other input range. This is completely dependent on the application, and resolving this final stage transfer function is entirely up to a control engineer and not part of this software.

Example for RTPIDControl.

The example below is extracted from the PIDControlTuner example. The code really needs to be studied in the context of the complete program so study that example program instead of the abbreviated code below.




```
RTProcessVar []ProportionalControl= new RTProcessVar[8];
RTProcessVar []IntegralControl= new RTProcessVar[8];
RTProcessVar []DerivativeControl= new RTProcessVar[8];
RTProcessVar []ControlSetpoints= new RTProcessVar[8];
RTProcessVar []ControlTrackBarOutputs= new RTProcessVar[8];
RTProcessVar []ControlOutputs= new RTProcessVar[8];
RTPIDControl []PIDControlLoops= new RTPIDControl[8];
.
.
for (int i=0; i <16; i++)
{
.
.
.
// Note derivative value is saled down by 100x
               // Note derivative value is saled down by 100x
    PIDControlLoops[i] =  new RTPIDControl(ControlSetpoints[i].getCurrentValue(),
        pidSteadyState, ProportionalControl[i].getCurrentValue(),
        IntegralControl[i].getCurrentValue(),
        DerivativeControl[i].getCurrentValue()/ 100.0,
        sampleper/60.0, filterConstant);
}


.
.
.
void  CalculatePIDValues ()
{
    double rMeas = 0.0;
    double rSetpoint = 0.0;
    double rOutput = 0.0;
    for (int i = 0; i < 8; i++)
    {
        // Not calculating the PID value will prevent integral errors from
        // continuing to be added to error sum
        // simulate measurement
        rOutput = ControlOutputs[i].getCurrentValue();
        rMeas = ProcessModel (i, rOutput);
        PIDProcessItems[i].setCurrentValue(rMeas);
        if (autoModeEnable[i])
        {
```

```
            rSetpoint = ControlSetpoints[i].getCurrentValue();
            rOutput = PIDControlLoops[i].calcPID(rMeas, rSetpoint);
            ControlOutputs[i].setCurrentValue(rOutput);
        }
    }
    outputControlTrackBar.setRTValue(
        ControlOutputs[currentTuningChannel].getCurrentValue());
}
```

16. Zooming Real-Time Data

ChartZoom

Zooming is the interactive re-scaling of a charts physical coordinate system and the related axes based on limits defined by clicking and dragging a mouse inside the current graph window. A typical use of zooming is in applications where the initial chart displays a large number of data points. The user interacts with the chart, defining smaller and smaller zoom rectangles, zeroing in on the region of interest. The final chart displays axis limits that have a very small range compared to the range of the original, un-zoomed, chart.

The **ChartZoom** class found in the **QCChart2D** software is used for zooming of **RTProcessVar** historical data. The features of this class include:

- Automatic recalculation of axis properties for tick mark spacing and axis labels..
- Zooming of time coordinates with smooth transitions between major scale changes: years->months->weeks->days->hours->minutes->seconds.
- Zooming of time coordinates that use a 5-day week and a non-24 hour day.
- Simultaneous zooming of an unlimited number of x- and y-coordinate systems and axes (super zooming).
- The user can recover previous zoom levels using a zoom stack.
- User-definable zoom limits prevent numeric under and overflows

The **Real-Time Graphics Tools for Java** software was designed to allow zooming of real-time data, while the data is being collected. Real-time data values are updated, using the **RTProcessVar** class, asynchronous to the update of the screen. The current graphics display can be historical data from the same **RTProcessVar** object, or it can be based on an entirely different **RTProcessVar** object. It is therefore possible to zoom back to the beginning of an **RTProcessVar** objects historical buffer, without affecting current data collection. At any time the graph return to a view that includes the most current information.

When you want to zoom or pan backwards into the historical buffer of the **RTProcessVar** object, first you must disable the **RTScrollFrame** updates. Since the **RTScrollFrame** will attempt to always graph the most recent data values, you don't want it interfering with a zoom or a pan which explicitly does NOT want the most recent

values displayed. Disable the **RTScrollFrame** updates using the **RTScrollFrame.setChartObjEnable** method: set it to ChartConstants.OBJECT_DISABLE. When you want to start scrolling again set it to ChartConstants.OBJECT_ENABLE.

Simple Zooming of a single channel scroll frame

Class ChartZoom

ChartMouseListener
|
+--ChartZoom

The **ChartZoom** class implements the Java **MouseInputListener** interface. It implements and uses the **MouseInputListener** mouse events: mouseMoved, mouseDragged, mousePressed, mouseReleased, mouseClicked and mouseEntered. The default operation of the **ChartZoom** class starts the zoom operation on the mousePressed event; it draws the zoom rectangle using the XOR drawing mode during the mouseDragged event; and terminates the zoom operation on the mouse released event. During the mouse released event, the zoom rectangle is converted from Java user units into the chart physical coordinates and this information is stored and optionally used to rescale the chart scale and all axis objects that reference the chart scale. If four axis objects reference a single chart scale, for example when axes bound a chart on all for sides, all four axes re-scale to match the new chart scale.

ChartZoom constructor

The constructor below creates a zoom object for a single chart coordinate system.

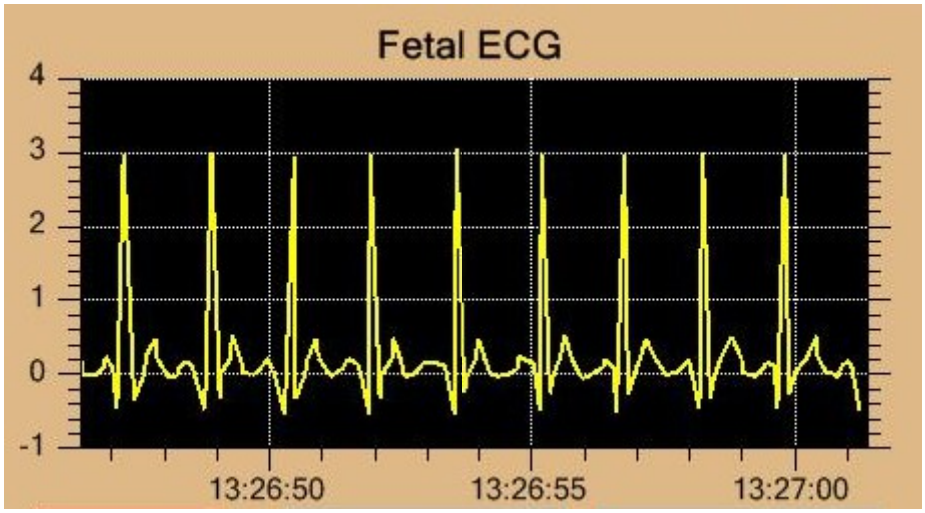
```
public ChartZoom(
   ChartView component,
   PhysicalCoordinates transform,
   boolean brescale
);
```

component	A reference to the **ChartView** object that the chart is placed in.
transform	The **PhysicalCoordinates** object associated with the scale being zoomed.
brescale	True designates that the scale should be re-scaled, once the final zoom rectangle is ascertained.

Once created, enable the **ChartZoom** by calling the **ChartZoom.addZoomListener** method. This adds the **ChartZoom** object as a **MouseListener** and a **MouseMotionListener** to the **ChartView** object. Temporarily disable the object using the **ChartZoom.setZoomEnable (false)** method. Call the **ChartZoom.removeZoomListener** method to remove the object as a mouse listener for the chart view.

Retrieve the physical coordinates of the zoom rectangle using the **ChartZoom** getZoomMin and getZoomMax methods. Restrict zooming in the x- or y-direction using the setZoomXEnable and setZoomYEnable methods. Set the rounding mode associated with rescale operations using the setZoomXRoundMode and setZoomYRoundMode methods. Call the **ChartZoom.popZoomStack** method at any time and the chart scale reverts to the minimum and maximum values of the previous zoom operation. Repeated calls to the popZoomStack method return the chart scale is to its original condition, after which the popZoomStack method has no effect.

Simple zoom example (Extracted from the FetalMonitor example)
In this example, a new class derives from the **ChartZoom** class and the mousePressed event overridden. The event invokes the **PopZoomStack** method. Otherwise, the default operation of the **ChartZoom** class controls everything else.




```
class ZoomWithStack extends ChartZoom
{

    public ZoomWithStack(ChartView component,
        TimeCoordinates transforms,  boolean brescale)
    {
        super( component,  transforms, brescale);
        }
    public void mouseClicked (MouseEvent event)
    {
      int mods;
      mods = event.getModifiers();
```

```
          if ((mods & buttonMask) != MouseEvent.BUTTON3_MASK)
            popZoomStack();
        }
    }
.
.
ZoomWithStack zoomObj;
.
.
zoomObj = new ZoomWithStack(chartVu, pTransform1, true);
zoomObj.setButtonMask(MouseEvent.BUTTON1_MASK);
zoomObj.setZoomYEnable(true);
zoomObj.setZoomXEnable(true);
zoomObj.setZoomXRoundMode(ChartConstants.AUTOAXES_FAR);
zoomObj.setZoomYRoundMode(ChartConstants.AUTOAXES_FAR);
zoomObj.setZoomRangeLimits(1000, 0.5);
zoomObj.addZoomListener();
zoomObj.setEnable(false);
.
.
.

private void zoomOn_Button_Click( MouseEvent e)
{
    // Change to display of all collected data


fetalHeartECGScrollFrame.setScrollScaleModeX(ChartConstants.RT_AUTOSCALE_X_MINMAX)
;
    // Look at updatecounter number of points, which is all of them
    fetalHeartECGScrollFrame.setMaxDisplayHistory(updatecounter);
    // Render graph based on new scale, showing all past data points
    this.updateDraw();
    // Now disable scroll frame
    fetalHeartECGScrollFrame.setChartObjEnable(ChartConstants.OBJECT_DISABLE);

    // Turn on zooming
    zoomObj.setEnable(true);
}

    private void zoomRestore_Button_Click( MouseEvent e)
    {
        RTControlButton button = (RTControlButton) e.getSource();
        // Turn off zooming
        zoomObj.setEnable(false);
        // Restore original y-scale values
        fetalHeartECGScrollFrame.getChartObjScale().setScaleStartY(-1.0);
        fetalHeartECGScrollFrame.getChartObjScale().setScaleStopY(4.0);
        // Re-establish scroll mode


fetalHeartECGScrollFrame.setScrollScaleModeX(ChartConstants.RT_FIXEDEXTENT_MOVINGS
TART_AUTOSCROLL);
        fetalHeartECGScrollFrame.setChartObjEnable(ChartConstants.OBJECT_ENABLE);
        // Render graph
        this.updateDraw();
    }
```

Super Zooming of multiple physical coordinate systems

The **ChartZoom** class also supports the zooming of multiple physical coordinate systems (*super zooming*). During the mouse released event, the zoom rectangle is converted from device units into the physical coordinates of each scale, and this information is used to re-scale each coordinate system, and the axis objects associated with them.

Use the constructor below in order to super zoom a chart that has multiple coordinate systems and axes.

ChartZoom constructor

```
public ChartZoom(
   ChartView component,
   PhysicalCoordinates[] transforms,
   boolean brescale
);
```

component	A reference to the **ChartView** object that the chart is placed in.
transforms	An array, size numtransforms, of the **PhysicalCoordinates** objects associated with the zoom operation.
brescale	True designates that the all of the scales should be re-scaled, once the final zoom rectangle is ascertained.

Once created, enable the **ChartZoom** by calling the **ChartZoom.addZoomListener** method. This adds the **ChartZoom** object as a **MouseListener** and a **MouseMotionListener** to the **ChartView** object. Temporarily disable the object using the **ChartZoom.setZoomEnable (false)** method. Call the **ChartZoom.removeZoomListener** method to remove the object as a mouse listener for the chart view.

Restrict zooming in the x- or y-direction using the setZoomXEnable and setZoomYEnable methods. Set the rounding mode associated with rescale operations using the setZoomXRoundMode and setZoomYRoundMode methods. Call the **ChartZoom.popZoomStack** method at any time and the chart scale reverts to the minimum and maximum values of the previous zoom operation. Repeated calls to the popZoomStack method return the chart scale is to its original condition, after which the popZoomStack method has no effect.


```
    class ZoomWithStack extends ChartZoom
    {

        public ZoomWithStack(ChartView component, TimeCoordinates transforms,
boolean brescale)
        {
            super( component,  transforms, brescale);
            }
        public void mouseClicked (MouseEvent event)
        {
          int mods;
          mods = event.getModifiers();
          if ((mods & buttonMask) != MouseEvent.BUTTON3_MASK)
            popZoomStack();
        }
    }.
.
ZoomWithStack zoomObj;
.
.
zoomObj = new ZoomWithStack(chartVu, pTransform1, true);
zoomObj.setButtonMask(MouseEvent.BUTTON1_MASK);
zoomObj.setZoomYEnable(true);
zoomObj.setZoomXEnable(true);
zoomObj.setZoomXRoundMode(ChartConstants.AUTOAXES_FAR);
zoomObj.setZoomYRoundMode(ChartConstants.AUTOAXES_FAR);
zoomObj.setZoomRangeLimits(1000, 1.0);
zoomObj.addZoomListener();
zoomObj.setEnable(false);.
.
.

private void zoomOn_Button_Click(MouseEvent e)
{
    // Change to display of all collected data
    scrollFrame1.setScrollScaleModeX(ChartConstants.RT_AUTOSCALE_X_MINMAX);
    // Look at updatecounter number of points, which is all of them
    scrollFrame1.setMaxDisplayHistory ( updatecounter);
    // Render graph based on new scale

    // Change to display of all collected data
    scrollFrame2.setScrollScaleModeX ( ChartConstants.RT_AUTOSCALE_X_MINMAX);
    // Look at updatecounter number of points, which is all of them
    scrollFrame2.setMaxDisplayHistory ( updatecounter);

// Update first, to display all historical information,
// then disable to allow for zooming.
    this.updateDraw();

    scrollFrame2.setChartObjEnable( ChartConstants.OBJECT_DISABLE);
    scrollFrame1.setChartObjEnable( ChartConstants.OBJECT_DISABLE);
    // Turn on zooming
    zoomObj.setEnable(true);
}
```

```
private void zoomRestore_Button_Click(MouseEvent e)
{

    RTControlButton button = (RTControlButton) e.getSource();
    // Turn off zooming
    zoomObj.setEnable(false);
    // Re-establish scroll mode


scrollFrame1.setScrollScaleModeX(ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCR
OLL);
    scrollFrame1.setChartObjEnable( ChartConstants.OBJECT_ENABLE);
    // Re-establish scroll mode
    scrollFrame2.setScrollScaleModeX(
    ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);
    scrollFrame2.setChartObjEnable( ChartConstants.OBJECT_ENABLE);

    // Render graph
    this.updateDraw();
}
```

Limiting the Zoom Range

A zoom window needs to have zoom limits placed on the minimum allowable zoom range for the x- and y-coordinates. Unrestricted or infinite zooming can result in numeric under and overflows. The default minimum allowable range resulting from a zoom operation is 1/1000 of the original coordinate range. Change this value using the **ChartZoom.setZoomRangeLimitsRatio** method. The minimum allowable range for this value is approximately 1.0e-9. Another way to set the minimum allowable range is to specify explicit values for the x- and y-range using the **ChartZoom.setZoomRangeLimits** method. Specify the minimum allowable zoom range for a time axis in milliseconds, for example **ChartZoom.setZoomRangeLimits(new ChartDimension(1000, 0.01))** sets the minimum zoom range for the time axis to 1 second and for the y-axis to 0.01. The utility method **ChartCalendar.getCalendarWidthValue** is useful for calculating the milliseconds for any time base and any number of units. The code below sets a minimum zoom range of 45 minutes.

```
double minZoomTimeRange  =
         ChartCalendar.getCalendarWidthValue(ChartConstants.MINUTE, 45);
double minZoomYRange = 0.01;
ChartDimension zoomLimits = new ChartDimension (minZoomTimeRange, minZoomYRange);
zoomObj.setZoomRangeLimits(zoomLimits);
```

17. Miscellaneous Shape Drawing

Com.quinncurtis.chart2djava.GraphObj
RT3DFrame
Com.quinncurtis.chart2djava.GraphObj
RTRoundedRectangle2D
Com.quinncurtis.chart2djava.GraphObj
RTGenShape

Often the look and feel of a real-time display is greatly enhanced with the addition of a few simple circular and rectangular drawing shapes. All of the example programs use these shapes, either directly, or indirectly as the 3D border element of the **RTPanelMeter** class. The chapter discusses how to explicitly add these objects to your program

3D Borders and Background Frames

Class RT3DFrame

Com.quinncurtis.chart2djava.GraphObj
RT3DFrame

This class is used to draw 3D borders and provide the background for many of the other graph objects, most noticeably the **RTPanelMeter** classes. It can also be used directly in your program to provide 3D frames the visually group objects together in a faceplate format.

RT3DFrame constructors

```
public RT3DFrame(
   PhysicalCoordinates transform,
   ChartRectangle2D rect,
   ChartAttribute attrib,
   int postype
);
```

Parameters

transform
Places the **RT3DFrame** object in the coordinate system defined by transform.

rect
Specifies the position and size of the frame.

attrib
Specifies the attributes (line and fill color) for the frame.

postype
Specifies the positioning coordinate system.

Selected Public Instance Properties*

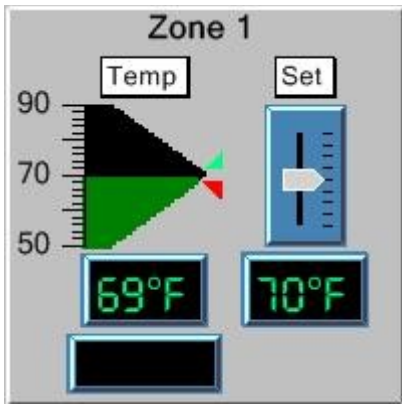
* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. value := **getBoundingBox**();

BoundingBox (inherited from **GraphObj**)	Returns the bounding box for the chart object. Not all chart objects have bounding boxes. Be sure and check for null.
ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a **ChartAttribute** object.
FillColor (inherited from **GraphObj**)	Sets the fill color for the chart object.
FrameRect	Set/Get the **ChartRectangle2D** object used to define the position and size of the 3D frame.
LightMode	Set/Get the apparent direction of the light source used to highlight the 3D frame. Use one of the direction constants. LIGHT_UPPER_LEFT, LIGHT_UPPER_RIGHT, LIGHT_LOWER_LEFT, LIGHT_LOWER_RIGHT , LIGHT_STRAIGHT_ON , LIGHT_NONE, OUTSET_3D_LOOK, INSET_3D_LOOK.
LineColor (inherited from **GraphObj**)	Sets the line color for the chart object.
LineStyle (inherited from **GraphObj**)	Sets the line style for the chart object.
LineWidth (inherited from **GraphObj**)	Sets the line width for the chart object.
PositionType (inherited from **GraphObj**)	Get/Sets the current position type.

A complete listing of **RT3DFrame** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

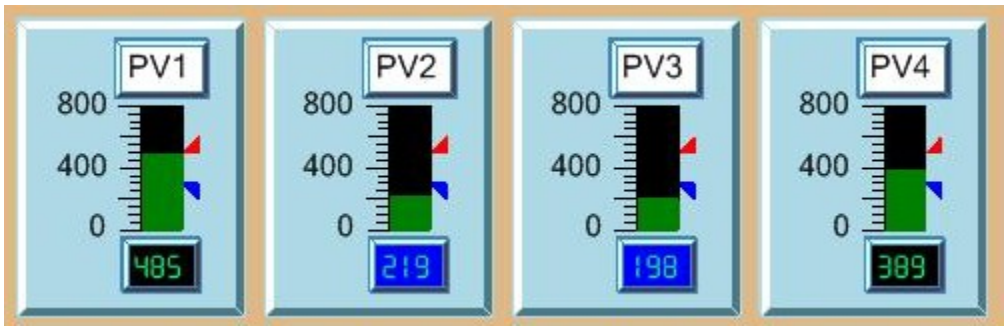
Example for drawing RT3DFrame objects

The example below, extracted from the HomeAutomation example, file ThermostatControl, draws an **RT3DFrame** object the size of the entire control area. Since each separate control in the example has a similar **RT3DFrame** background, it provides a visual grouping of the objects in each control.




```
private void InitializeBackgroundPanel()
{
    ChartView chartVu = this;
    Font theFont = font16Bold;
    ChartRectangle2D normrect = new ChartRectangle2D(0.0, 0.0, 1.0, 1.0);
    CartesianCoordinates pTransform1 =    new CartesianCoordinates();
    RT3DFrame frame3d = new RT3DFrame(pTransform1, normrect,
    facePlateAttrib, ChartObj.NORM_GRAPH_POS);
           chartVu.addChartObject(frame3d);
}
```

The example below, extracted from the ProcessMonitoring example, file ProcessMonitoring, method **InitializeTopBargraphs**, draws an **RT3DFrame** object as a backdrop for each of the barographs.




```
for (int i=0; i < PIDProcessItems.length; i++)
{
    row = i/4;
    col = i % 4;
    x1 = xoffset + col * faceplatewidthspacing;
    y1 = yoffset + row * faceplateheightspacing;
    x2 = x1+faceplatewidth;
```

```
    y2 = y1+faceplateheight;

    double mindisplayvalue = PIDProcessItems[i].getDefaultMinimumDisplayValue();
    double maxdisplayvalue = PIDProcessItems[i].getDefaultMaximumDisplayValue();

    pTransform1 = new CartesianCoordinates( 0.0, mindisplayvalue,
        1.0, maxdisplayvalue);

    ChartRectangle2D normrect = new ChartRectangle2D(x1, y1,
        faceplatewidth, faceplateheight);
    RT3DFrame frame3d = new RT3DFrame(pTransform1, normrect,
        rectattrib, ChartConstants.NORM_GRAPH_POS);
    chartVu.addChartObject(frame3d);
}
```

Rounded Rectangles

Class RTRoundedRectangle2D

Com.quinncurtis.chart2djava.GraphObj
RTRoundedRectangle2D

Rounded rectangles are just that, rectangles that have rounded corners.

RTRoundedRectangle2D constructors

```
public RTRoundedRectangle2D(
   PhysicalCoordinates transform,
   ChartRectangle2D r,
   double corner,
   int postype
);
```

Parameters

transform

Places the **RTRoundedRectangle2D** object in the coordinate system defined by transform.

r

The size and position of the rectangle.

corner

The radius of the rectangle corners.

postype

The coordinate system the rectangle is defined in. Use one of the coordinate system constants: DEV_POS, PHYS_POS, NORM_GRAPH_POS, NORM_PLOT_POS.

Selected Public Instance Properties*

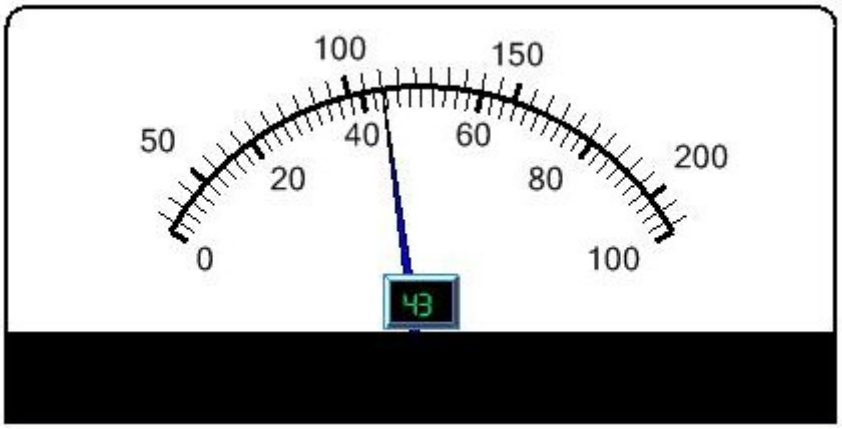
* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setCornerRadius**(value) and value := **getCornerRadius**();

CornerRadius	Get/Set the corner radius of the rounded rectangle.
FillColor (inherited from **GraphObj**)	Sets the fill color for the chart object.
Height	Get/Set the height of the rectangle.
LineColor (inherited from **GraphObj**)	Sets the line color for the chart object.
LineStyle (inherited from **GraphObj**)	Sets the line style for the chart object.
LineWidth (inherited from **GraphObj**)	Sets the line width for the chart object.
PositionType (inherited from **GraphObj**)	Get/Sets the current position type.
Width	Get/Set the width of the rectangle.
X	Get/Set the x-value of the rectangle.
Y	Get/Set the y-value of the rectangle.
ZOrder (inherited from **GraphObj**)	Sets the z-order of the object in the chart. Every object has a z-order value. Each z-order value does NOT have to be unique. If z-order sorting is turned on in the **ChartView** object, objects are sorted by z-order before they are drawn.

A complete listing of **RTRoundedRectangle2D** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for drawing RTRoundedRectangle2D objects

The example below, extracted from the MeterDemo example, file NeedleMeter, method **InitializeMeter2**., draws a large rectangle with rounded corners as the frame of the meter, and a smaller, filled rectangle at the bottom.




```
RTRoundedRectangle2D rr =
    new RTRoundedRectangle2D(meterframe1, 0.25, 0.01, 0.45, 0.43, 0.01,
     ChartConstants.NORM_GRAPH_POS);
```

```
ChartAttribute backattrib =
    new ChartAttribute(Color.black,2,ChartConstants.LS_SOLID,Color.white);
rr.setChartObjAttributes(backattrib);
rr.setChartObjClipping(ChartConstants.NO_CLIPPING);
chartVu.addChartObject(rr);
RTRoundedRectangle2D rrr =
new RTRoundedRectangle2D(meterframe1, 0.25, 0.35, 0.45, 0.09, 0.0,
    ChartConstants.NORM_GRAPH_POS);
ChartAttribute backattrib2 = new
    ChartAttribute(Color.black,2,ChartConstants.LS_SOLID,Color.black);
rrr.setChartObjAttributes(backattrib2);
rrr.setZOrder(60);
rrr.setChartObjClipping(ChartConstants.NO_CLIPPING);
chartVu.addChartObject(rrr);
```

General Shapes

Class RTGenShape

Com.quinncurtis.chart2djava.GraphObj
RTGenShape

This class is used to draw filled and unfilled rectangles, rectangles with rounded corners, general ellipses and aspect ratio corrected circles. These shapes can be used by the programmer to add visual enhancements to graphs.

RTGenShape constructors

```
public RTGenShape(
   PhysicalCoordinates transform,
   ChartRectangle2D rect,
   double corner,
   int shape,
   int postype
);
```

Parameters

transform
: The coordinate system for the new **RTGenShape** object.

rect
: The source rectangle.

corner
: The corner radius of the rounded rectangle.

shape
: The shape of the **RTGenShape** object. Use one of the generalized shape constants: RT_SHAPE_RECTANGLE, RT_SHAPE_ROUNDEDRECTANGLE, RT_SHAPE_ELLIPSE.

postype
: Specifies what coordinate system the coordinates reference. Use one of the position type constants: DEV_POS, PHYS_POS, POLAR_POS, NORM_GRAPH_POS, NORM_PLOT_POS.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

AspectRatioCorrection	Get/Set the aspect ratio correction mode for the RT_SHAPE_ELLIPSE shape: NO_ASPECT_RATIO_CORRECTION, FIXED_X_ASPECT_RATIO_CORRECTION, FIXED_Y_ASPECT_RATIO_CORRECTION.
ChartObjAttributes (inherited from **GraphObj**)	Sets the attributes for a chart object using a **ChartAttribute** object.
CornerRadius	Get/Set the corner radius of the rounded rectangle.
FillColor (inherited from **GraphObj**)	Sets the fill color for the chart object.
GenShape	Get/Set the shape control property genShape. Use one of the generalized shape constants: RT_SHAPE_RECTANGLE, RT_SHAPE_ROUNDEDRECTANGLE, RT_SHAPE_ELLIPSE.
Height	Get/Set the height of the shape rectangle.
LineColor (inherited from **GraphObj**)	Sets the line color for the chart object.
LineStyle (inherited from **GraphObj**)	Sets the line style for the chart object.
LineWidth (inherited from **GraphObj**)	Sets the line width for the chart object.
PositionType (inherited from **GraphObj**)	Get/Sets the current position type.
ShapeRect	Get/Set the rectangle control the size and position of the object.
Width	Get/Set the width of the rectangle.
X	Get/Set the x-value of the shape rectangle.
Y	Get/Set the y-value of the shape rectangle.
ZOrder (inherited from **GraphObj**)	Sets the z-order of the object in the chart. Every object has a z-order value. Each z-order value does NOT have to be unique. If z-order sorting is turned on in the **ChartView** object, objects are sorted by z-order before they are drawn.

A complete listing of **RTGenShape** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

Example for drawing RTGenShape objects

The example below, extracted from the AutoInstrumentPanel example, file AutoInstrumentPanel, method **InitializeClock**, draws a circle around the borders of the clock.

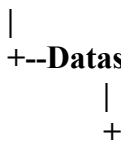
```
chartAttribute attrib2 = new ChartAttribute (Color.gray,
5,ChartConstants.LS_SOLID,
Color.white);
ChartRectangle2D shaperect = new ChartRectangle2D(0.8, 0.025, 0.19, 0.25);
RTGenShape genshape = new RTGenShape(meterframe,
shaperect,0.0,ChartConstants.RT_SHAPE_ELLIPSE, ChartConstants.NORM_GRAPH_POS);
genshape.setChartObjAttributes(attrib2);
chartVu.addChartObject(genshape);
```

18. Process Variable Viewer

RTProcessVarViewer

The **RTProcessVarViewer** class extends the QCChart2D **DatasetViewer** class so that it can display the historical datasets stored in the **RTProcessVar** objects. The **RTProcessVarViewer** can be updated in real-time, and synchronized to the chart, so that scrolling of the **RTProcessVarViewer** can scroll the chart.

Class RTProcessVarViewer

ChartView

+--DatasetViewer

+--RTProcessVarViewer

The **RTProcessVarViewer** is a **ChartView** derived object and as such is an independent **JPanel** derived object. Use it to view one or more **RTProcessVar** objects in a real-time display. Since it is usually not possible or practical to display the entire dataset, the **RTProcessVarViewer** windows a rectangular section of the dataset for display. Scroll bars are used to scroll the rows and columns of the dataset. The **RTProcessVarViewer** constructor defines the size, position, source matrix, the number of rows and columns of the **RTProcessVarViewer** grid, and the starting position of the **RTProcessVarViewer** scrollbar.

RTProcessVarViewer constructor

```
public RTProcessVarViewer(
    ChartView chartvu,
    PhysicalCoordinates transform,
    ChartRectangle2D  posrect,
    RTProcessVar pv,
    int rows,
    int cols,
    int start
)
```

chartvu The **ChartView** object the **DatasetViewer** is placed in.

transform	The coordinate system the **DatasetViewer** is placed in.
posrect	A positioning rectangle (using normalized chart coordinates) for the dataset viewer, use null if not used.
pv	The initial process variable.
rows	Number of rows to display
cols	Number of columns to display.
start	Starting column of the dataset viewer.

Set unique fonts for the column headers, row headers and grid cells using the setColumnHeaderFont, setRowHearderFont and setGridCellFont methods.

Turn on the edit feature of the grid cells using the setEnableEdit property. Turn on the striped background color of the grid cells using the setUseStripedGridBackground methods.

Foreground and background attributes of the column headers, row headers and grid cells can be set using the setColumnHeaderAttribute, setRowHeaderAttribute, setGridAttribute, and setAltGridAttribute methods.

You can add multiple **RTProcessVar** objects toa **RTProcessVarViewer** using the RTProcessVarViewer.addProcessVar method. When adding additional process variables, it only adds the y-values of the dataset. It is assumed the x-values of the datasets are the same; otherwise, the columns would lose synchronization.

The row header string for the first grid row, the x-values, is picked up from the first dataset's XString property. If that is null, "X-Values" is displayed for numeric x-values, and "Time" for time-based x-values. Subsequent row header strings, for the y-values, are picked up from the main title string of each associated dataset. In the case of group datasets with multiple y-values for each x-value, row header strings are picked up from the datasets GroupStrings property, which stores one string for each group in the dataset.

You can change the default orientation of the **RTProcessVarViewer** by calling a version of the **RTProcessVarViewer** constructor that has an orientation property as the last parameter. See the ProcessVarTables.VerticalScrollApplicationUserControl1.cs for an example.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

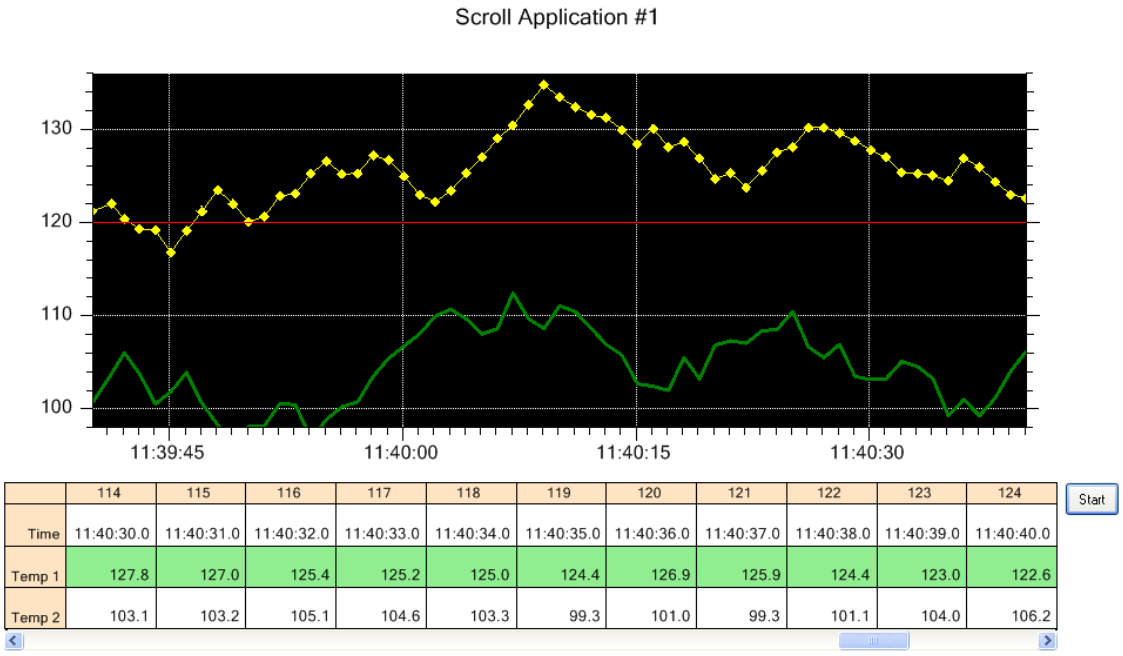
AltGridAttribute	(Inherited from DataGridBase.)
AutoRedrawTable	Set to true and the table will redraw using the current data associated with the update of th RTProcessVar.
ColumnHeaderAttribute	(Inherited from DataGridBase.)

ColumnHeaderFont	(Inherited from DatasetViewer.)
ColumnHeads	(Inherited from DataGridBase.)
DataArray	(Inherited from DatasetViewer.)
DoubleBufferEnable	(Inherited from ChartView.)
DrawEnable	(Inherited from ChartView.)
GridAttribute	(Inherited from DataGridBase.)
GridCellFont	(Inherited from DatasetViewer.)
Height	Get/Set the height of the control.
HorizontalGroupPlot	(Inherited from DatasetViewer.)
HScrollBar1	(Inherited from DataGridBase.)
Left	Get/Set the distance, in pixels, between the left edge of the control and the left edge of its container's client area.
NumCols	(Inherited from DataGridBase.)
NumericFormat	(Inherited from DataGridBase.)
NumRows	(Inherited from DataGridBase.)
ParentChartView	(Inherited from DataGridBase.)
ParentTransform	(Inherited from DataGridBase.)
PreferredSize	(Inherited from ChartView.)
ResizeMode	(Inherited from ChartView.)
Right	Gets the distance, in pixels, between the right edge of the control and the left edge of its container's client area.
RowHeaderAttribute	(Inherited from DataGridBase.)
RowHeaderFont	(Inherited from DatasetViewer.)
RowHeads	(Inherited from DataGridBase.)
SmoothingMode	(Inherited from ChartView.)
SourceDataset	(Inherited from DatasetViewer.)
StartCol	(Inherited from DataGridBase.)
StartRow	(Inherited from DataGridBase.)
SyncChart	(Inherited from DatasetViewer.)
TableGreenBarFlag	(Inherited from DataGridBase.)
TableStartPosX	(Inherited from DataGridBase.)
TableStartPosY	(Inherited from DataGridBase.)
TableStopPosX	(Inherited from DataGridBase.)
TableStopPosY	(Inherited from DataGridBase.)
TextRenderingHint	(Inherited from ChartView.)
Title	(Inherited from DataGridBase.)

Top	Get/Set the distance, in pixels, between the top edge of the control and the top edge of its container's client area.
TransformList	(Inherited from DataGridBase.)
UseStripedGridBackground	(Inherited from DataGridBase.)
Visible	Get/Set a value indicating whether the control is displayed.
VScrollBar1	(Inherited from DataGridBase.)
Width	Get/Set the width of the control.

A complete list of RTProcessVarViewer classes is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file

Simple RTProcessVarViewer example (extracted from the example program ProcessVarDataTables.ScrollApplicationUserControl1.cs)



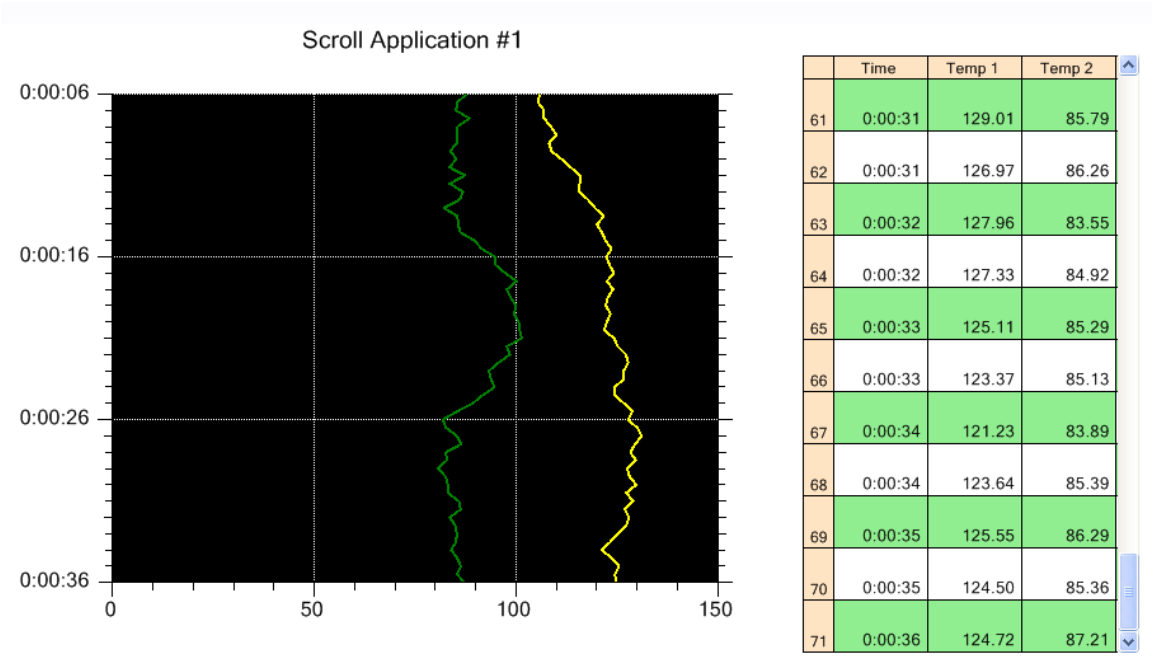
	114	115	116	117	118	119	120	121	122	123	124
Time	11:40:30.0	11:40:31.0	11:40:32.0	11:40:33.0	11:40:34.0	11:40:35.0	11:40:36.0	11:40:37.0	11:40:38.0	11:40:39.0	11:40:40.0
Temp 1	127.8	127.0	125.4	125.2	125.0	124.4	126.9	125.9	124.4	123.0	122.6
Temp 2	103.1	103.2	105.1	104.6	103.3	99.3	101.0	99.3	101.1	104.0	106.2

*A **RTProcessVarViewer** displaying two **RTProcessVar** objects*

```
this.setLayout(null);

ChartRectangle2D posrect = new ChartRectangle2D(0.05, 0.67, 0.87, 0.26);
int rows = 3, columns = 11, startindex = 0;
// Because the constructor is called, the RTProcessVarViewer is automatically
added to
// the chartVu.Controls object list.
```

```
rtProcessVarViewer1 = new RTProcessVarViewer(chartVu, pTransform1, posrect,
currentTemperature1, rows, columns, startindex);
rtProcessVarViewer1.setUseStripedGridBackground( true);
rtProcessVarViewer1.setGridCellFont( font12);
rtProcessVarViewer1.addProcessVar(currentTemperature2);
// .set custom decimal precision for each row
rtProcessVarViewer1.setFormatDecimalPos(0, 0);
rtProcessVarViewer1.setFormatDecimalPos(1, 1);
rtProcessVarViewer1.setFormatDecimalPos(2, 2);
```

Vertical Orientation DatasetViewer example (extracted from the example program ProcessVarDataTables.ElapsedTimeVerticalScrolling.)




```
this.setLayout(null);

ChartRectangle2D posrect = new ChartRectangle2D(0.7, 0.1, 0.29, 0.8);
int rows = 11, columns = 3, startindex = 0;
RTProcessVarViewer rtProcessVarViewer1 = new RTProcessVarViewer(chartVu,
pTransform1, posrect, currentTemperature1, rows, columns, 0, ChartObj.VERT_DIR);
rtProcessVarViewer1.setGridCellFont(font12);
rtProcessVarViewer1.addProcessVar(currentTemperature2);
```

19. Auto Indicator Classes

RTAutoBarIndicator
RTAutoMultiBarIndicator
RTAutoMeterIndicator
RTAutoClockIndicator
RTAutoDialIndicator
RTAutoScrollGraph
RTAutoPanelMeterIndicator

The auto-indicator classes are designed to simplify the creation of real-time displays. Each class encapsulates a collection of objects need to build a complete real-time indicator object: bar indicators, meters, dials, clocks and scrolling graphs.

There are seven self containted auto-indicator classes: single channel bar indicator, multi-channel bar indicator, meters, dials, clocks, panel meter, and scrolling graphs. The **ChartView** class is the base class for the auto-indicator classes. Each indicator is placed in its own **ChartView** derived window, along with all other objects typically associated the indicator (axes, labels, process variables, alarms, titles, etc.). Since **ChartView** is derived from **JPanel**, you can place as many auto-indicator classes on a form as you want.

Single Channel Bar Indicator

Class RTAutoBarIndicator

JPanel
ChartView
RTAutoIndicator
RTAutoBarIndicator

The **RTAutoBarIndicator** combines a **RTBarIndicator** object with other objects needed to create a self-contained bargraph display. These other objects include a **RTProcessVar** variable, axes, axis labels, title string, units string, alarm indicators, and panel meters used in the display of the bar graphs numeric value, tag name, and alarm status. Since it contains a single **RTProcessVar** object, it displays a single channel of data.

RTAutoBarIndicator constructors

Since the **RTAutoBarIndicator** is designed to be dropped on a form, only a default constructor is used. The indicator is customized using public properties.

```
public RTAutoBarIndicator ();
```

A couple of methods are used to initialize the bar graph after instantiation, InitBarIndicator and initStrings.

The initBarIndicator method initialized the orientation of the bars, the format of the bar graph, and the bar color.

Method initBarIndicator

```
public void initBarIndicator(
    int orientation,
    int bargraphformat,
    Color colr
)
```

Parameters

orientation

Specifies the orientation of the chart (ChartObj.VERT_DIR or ChartObj.HORIZ_DIR

bargraphformat

Specifies the bar graph format (0..3).

colr

The color of the bar.

The initStrings method initialized the tag and units strings.

Method initStrings

```
public void initStrings(
    string title,
    string units
)
```

Use the updateIndicator method to update the bar indicator with new data.

Method updateIndicator

```
public void updateIndicator(
    double value,
    boolean updatedraw
)
```

Parameters

value

Update the indicator channel with this value.

updatedraw

True and the indicator is immediately updated.

Selected Public Instance Properties*

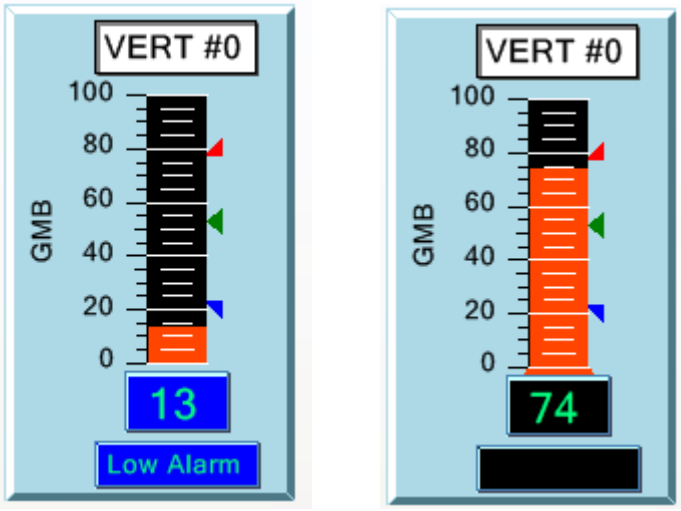
* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

Name	Description
AlarmIndicator	Get a reference to the RTAlarmIndicator object
AlarmPanelMeter	Get a reference to the RTAlarmPanelMeter object (Inherited from RTAutoIndicator.)
BarAttributes	Sets the line color for the chart object.
BarDataValue	Get the numeric label template object used to place numeric values on the bars.
BarEndBulb	Set/Get to true for a bar end bulb.
BarFillColor	Set/Get the fill color for the chart object.
BarLineWidth	Set/Get the line width for the chart object.
BarOrientation	Get/Set the orientation of the chart.
BarPlot	Get a reference to the RTBarIndicator object.
BarWidth	Set/Get the bar width.
BarWidthPixels	Set/Get to the pixel width of the bar in the bar plot.
CoordinateSystem	Get the coordinate system object for the indicator. (Inherited from RTAutoIndicator.)
FaceplateBackground	Set to true to show 3D faceplate (Inherited from RTAutoIndicator.)
GraphBackground	Get the graph background object. (Inherited from RTAutoIndicator.)
GraphBorder	Get the default graph border for the chart. (Inherited from RTAutoIndicator.)
GraphFormat	Get/Set any an indicator format, is supported (Inherited from RTAutoIndicator.)
Height	Set/Get the height of the control.
HighAlarm	Get the most recent high RTAlarm object (Inherited from RTAutoIndicator.)
InteriorAxis	Set/Get to true and an interior axis is drawn
LowAlarm	Get the most recent low RTAlarm object

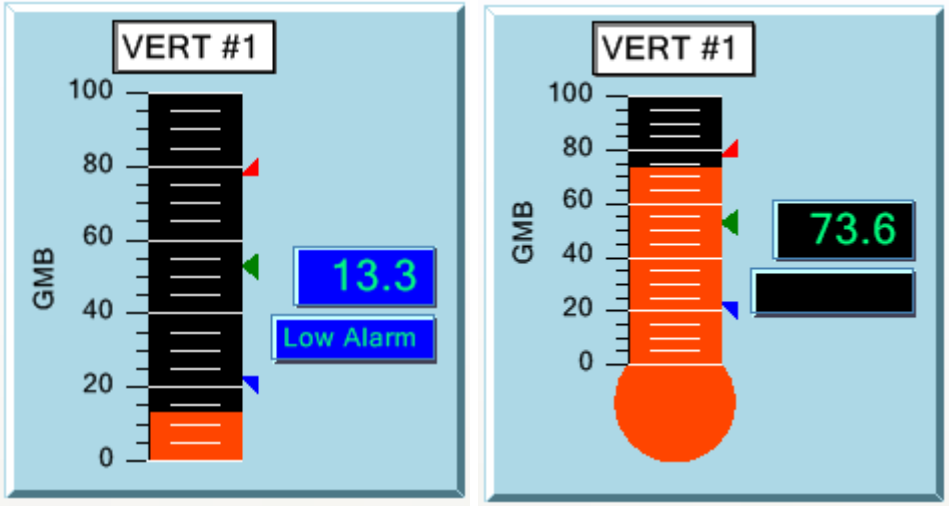
	(Inherited from RTAutoIndicator.)
MainTitle	Get/Set the tag string (Inherited from RTAutoIndicator.)
MaxIndicatorValue	The maximum value for the indicator. (Inherited from RTAutoIndicator.)
MinIndicatorValue	The minimum value for the indicator. (Inherited from RTAutoIndicator.)
NumericPanelMeter	Get a reference to the RTNumericPanelMeter object (Inherited from RTAutoIndicator.)
PlotAttrib	Get the ChartAttribute object associated with the plot.
PlotBackground	Get the plot background object. (Inherited from RTAutoIndicator.)
PreferredSize	Set/Get the preferred size of the control (Inherited from ChartView.) Established base size for font scaling.
ProcessVariable	Get most recently created RTProcessVar. (Inherited from RTAutoIndicator.)
RenderingMode	(Inherited from ChartView.)
ResizeMode	(Inherited from ChartView.)
SetpointAlarm	Get the most recent setpoint RTAlarm object (Inherited from RTAutoIndicator.)
TagPanelMeter	Get a reference to the tag panel meter object (Inherited from RTAutoIndicator.)
TagString	Get/Set the tag string (Inherited from RTAutoIndicator.)
UnitsPanelMeter	Get a reference to the units string panel meter object (Inherited from RTAutoIndicator.)
UnitsString	Get/Set the units string (Inherited from RTAutoIndicator.)
Visible	Set/Get a value indicating whether the control is displayed.
Width	Set/Get the width of the control.
XAxis	Get the x-axis object.
XAxis2	Get the second x-axis object.
XAxisLab	Get the x-axis labels object.
XAxisTitle	Get the x-axis title object.
XGrid	Get the x-axis grid object.
YAxis	Get the y-axis object.
YAxis2	Get the second y-axis object.
YAxisLab	Get the y-axis labels object. Accessible only after BuildGrap
YAxisTitle	Get the y-axis title object.

A complete listing of **RTAutoBarIndicator** classes is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file

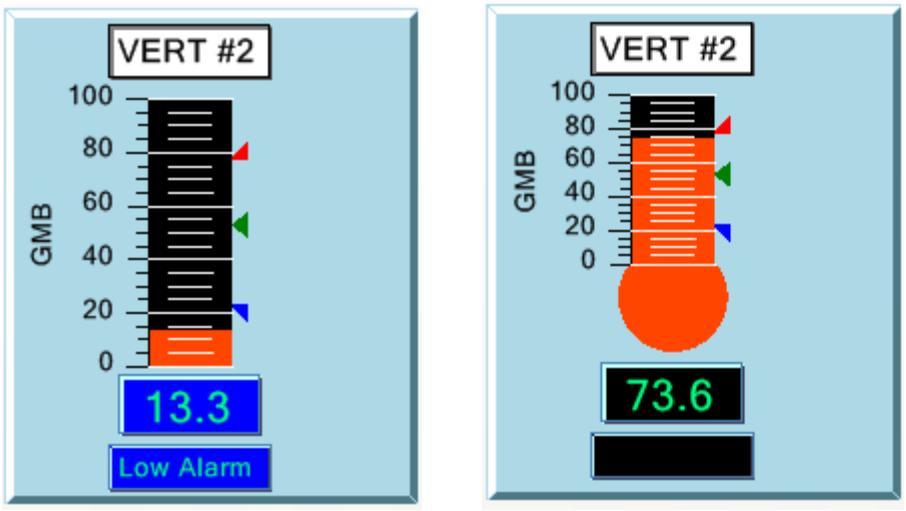
There are 8 different bar graph formats, four horizontal and four vertical. Use the setGraphFormat method (0..3) to set the format. Below you will find a brief description of the differences between the formats.



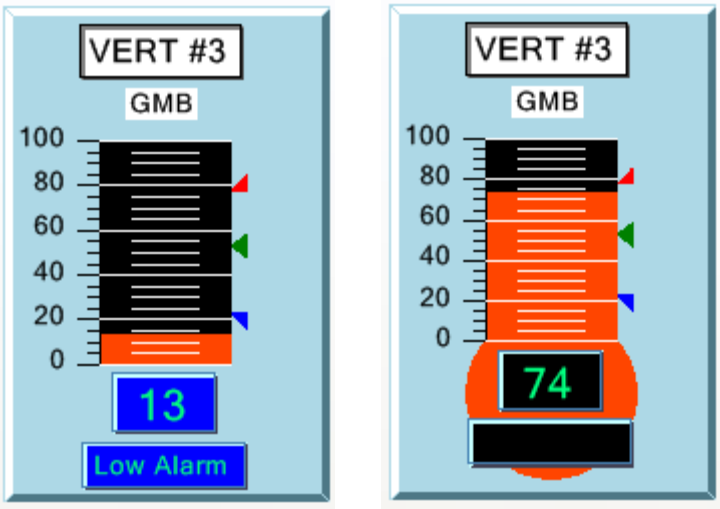
Panel meters above and below the bar for the tag name, numeric value, and alarm status. The scale units displayed vertically on the left. Turn the BarEndBulb property on and the numeric and alarm status panel meters will sit on top of the bar end bulb.



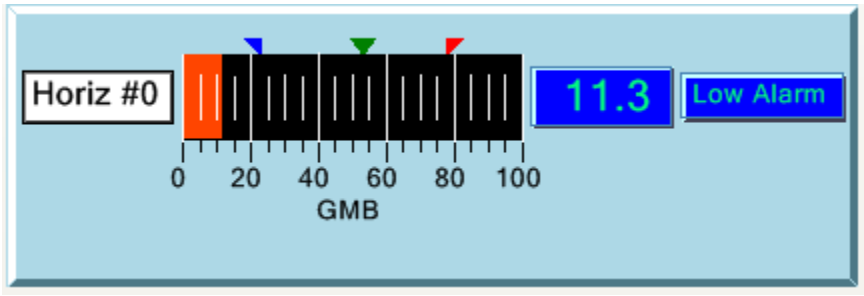
The tag panel meter on top, with the numeric value, and alarm status panel meters to the right. The scale units displayed vertically on the left. Turn on the BarEndBulb property and the bar indicator will shrink vertically in order fit the bulb in the indicator window.

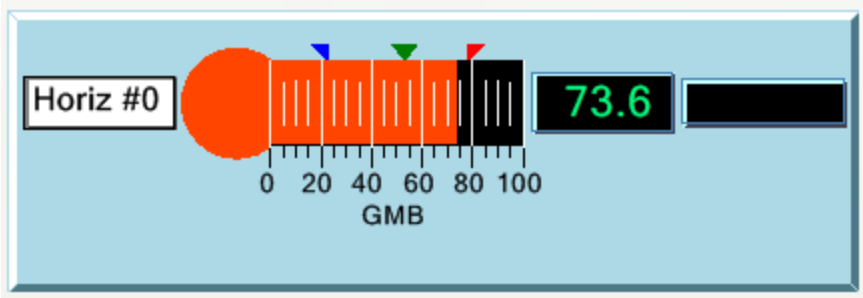


Similar to GraphFormat = 0, except that if the BarEndBulb property is turned on, the numeric and alarm status panel meters do not sit on top of the bar.

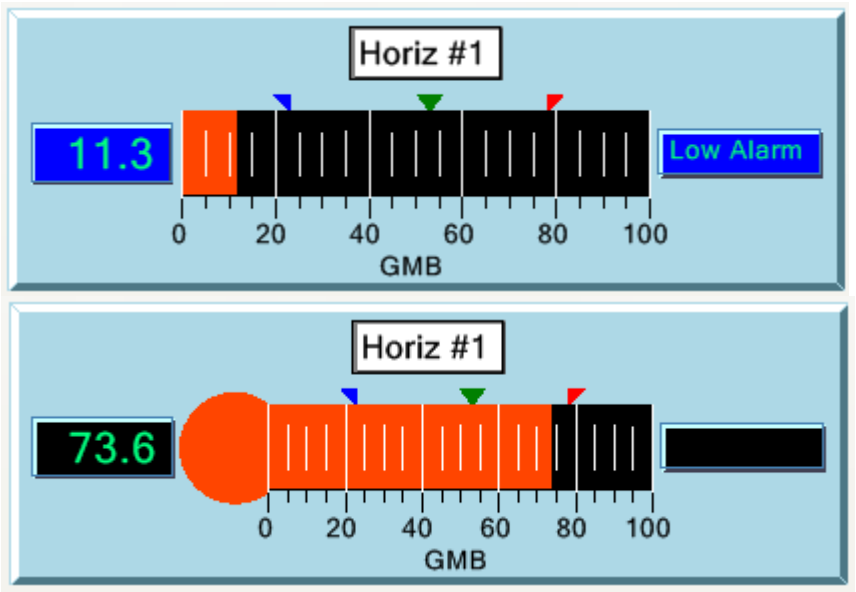


Panel meters above and below the bar for the tag name, numeric value, and alarm status. The scale units are displayed under the tag name. Turn the BarEndBulb property on and the numeric and alarm status panel meters will sit on top of the bar end bulb. The default width of this format is wider than GraphFormat = 0.

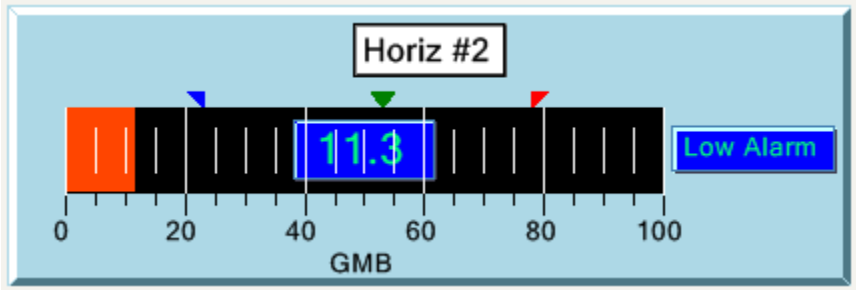


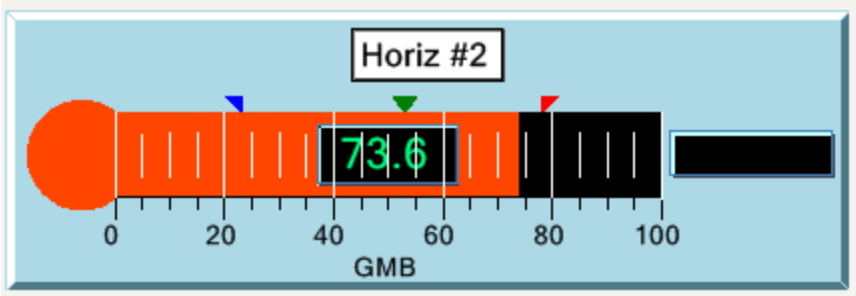


Panel meters to the left and right of the bar for the tag name, numeric value, and alarm status. The scale units displays horizontally under the scale. Turn the BarEndBulb property on and the bar indicator area shrinks horizontally in order to fit in the bulb without overlap.

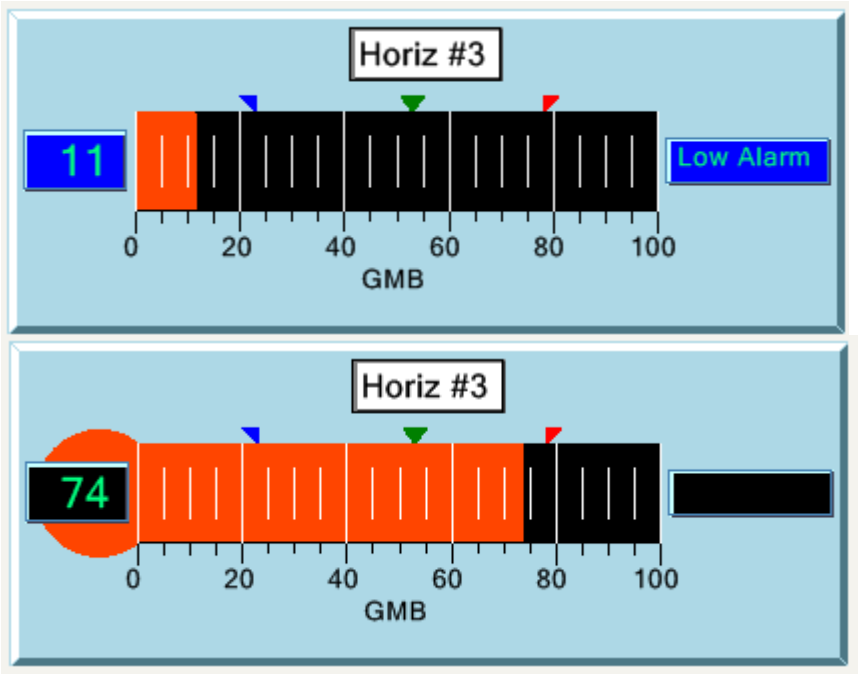


Panel meters to the left and right of the bar for the numeric value and and alarm status. A panel meter at the top for the tag name. The scale units displays horizontally under the scale. Turn the BarEndBulb property on and the bar indicator area shrinks horizontally in order to fit in the bulb without overlap.





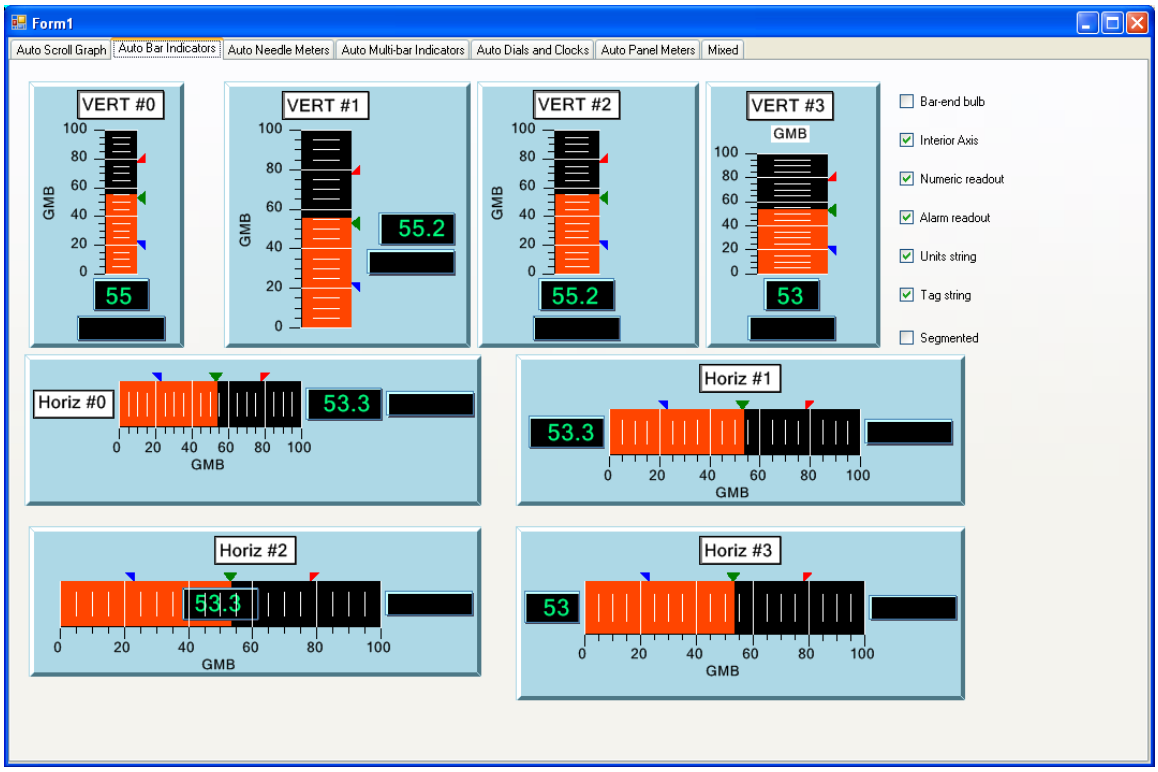
Panel meter right of the bar for the alarm status, with the numeric panel meter placed in the middle of the bar indicator area. A panel meter at the top for the tag name. The scale units displays horizontally under the scale. Turn the BarEndBulb property on and the bar indicator area shrinks horizontally in order to fit in the.



Simliar to GraphFormat = 1, except for the treatment of the bar end bulb. Panel meters to the left and right of the bar for the numeric value and and alarm status. A panel meter at the top for the tag name. The scale units displays horizontally under the scale. Turn the BarEndBulb property on and the numeric value sits on top of the bulb.

Example for initializing RTAutoBarIndicator objects

The example below, extracted from the AutoGraphDemos.AutoBarIndicators1 example, draws four vertical and four horizontal bargraphs.



Below you will find the code used to initialize the first of the bargraphs above, extracted from the AutoGraphDemos.AutoBarIndicators example program

```
private void InitializeBargraphs(boolean barbulb, boolean interioraxis,
            boolean numeric, boolean alarm, boolean units, boolean title,
            boolean segmented) {
        rtAutoBarIndicator1.setLocation(10, 10);
        rtAutoBarIndicator1.setSize(150, 250);
        rtAutoBarIndicator1.setPreferredSize(150, 250);
        rtAutoBarIndicator1.initBargraph(ChartObj.VERT_DIR, 0,
                ChartColors.ORANGERED);
        rtAutoBarIndicator1.initStrings("VERT #0", "GMB");
        rtAutoBarIndicator1.getLowAlarm().setAlarmLimitValue(23);
        rtAutoBarIndicator1.getHighAlarm().setAlarmLimitValue(78);
        rtAutoBarIndicator1.getSetpointAlarm().setAlarmLimitValue(53);
        rtAutoBarIndicator1.setMinIndicatorValue(0);
        rtAutoBarIndicator1.setMaxIndicatorValue(100);
        rtAutoBarIndicator1.getGraphBackground().setChartObjAttributes(
                new ChartAttribute(ChartColors.LIGHTBLUE, 5,
                        ChartConstants.LS_SOLID, ChartColors.LIGHTBLUE));
        rtAutoBarIndicator1.setFaceplateBackground(true);
        rtAutoBarIndicator1.setBarEndBulb(barbulb);
        rtAutoBarIndicator1.setInteriorAxis(interioraxis);
        rtAutoBarIndicator1.getNumericPanelMeter().setChartObjEnable(
                numeric ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoBarIndicator1.getNumericPanelMeter().getNumericTemplate()
                .setDecimalPos(0);
        rtAutoBarIndicator1.getAlarmPanelMeter().setChartObjEnable(
                alarm ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoBarIndicator1.getUnitsPanelMeter().setChartObjEnable(
```

```
            units ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoBarIndicator1.getYAxisTitle().setChartObjEnable(
            units ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoBarIndicator1.getTagPanelMeter().setChartObjEnable(
            title ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoBarIndicator1.getBarPlot().setIndicatorSubType(
            segmented ? ChartObj.RT_BAR_SEGMENTED_SUBTYPE
                : ChartObj.RT_BAR_SOLID_SUBTYPE);

.
.
.
```

Multi-Channel Bar Indicator

Class RTAutoMultiBarIndicator

JPanel
ChartView
RTAutoIndicator
RTAutoBarIndicator
RTAutoMultiBarIndicator

The **RTAutoMultiBarIndicator** combines a **RTMultiBarIndicator** object with other objects needed to create a self-contained multi-bargraph display. These other objects include an array of **RTProcessVar** variables, axes, axis labels, title string, units string, alarm indicators, and panel meters used in the display of the bar graphs numeric value, tag name, and alarm status. Since it contains an array of **RTProcessVar** objects, it can display one or more channels of data.

RTAutoMultiBarIndicator constructors

Since the **RTAutoMultiBarIndicator** is designed to be dropped on a form, only a default constructor is used. The indicator is customized using public properties.

```
public RTAutoMultiBarIndicator ();
```

A couple of methods are used to initialize the multi-bar graph after instantiation, **InitMultiBarIndicator** and initStrings.

The **initMultiBarIndicator** method initialized the orientation of the bars, the format of multi-bar graph, the princible bar color and the number of bars. If you want each bar to have a different color, call the **InitColors**(Color [] clrs) method, passing in one color for each bar.

Method initMultiBarIndicator

```
public void initMultiBarIndicator(
    int orientation,
    int bargraphformat,
    Color colr,
    int num
)
```

Parameters

orientation

Specifies the orientation of the chart (ChartObj.VERT_DIR or ChartObj.HORIZ_DIR

bargraphformat

Specifies the bar graph format.

colr

The color of the bars..

num

The number of bars in the mult-bargraph.

The initStrings method initialized the title, tags, and units strings.

Method initStrings

```
public void initStrings(
    string title,
    string units,
    string[] tags
)
```

Parameters

title

The title (or tag) string.

units

The units string.

tags

An array of the tag strings.

Use the **updateIndicator method to update the bar indicator with new data..**

Method updateIndicator

```
public void updateIndicator(
    double[] values,
    boolean updatedraw
)
```

Parameters

value

An array of new values, one for each channel of the indicator.

updatedraw

True and the indicator is immediately updated.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

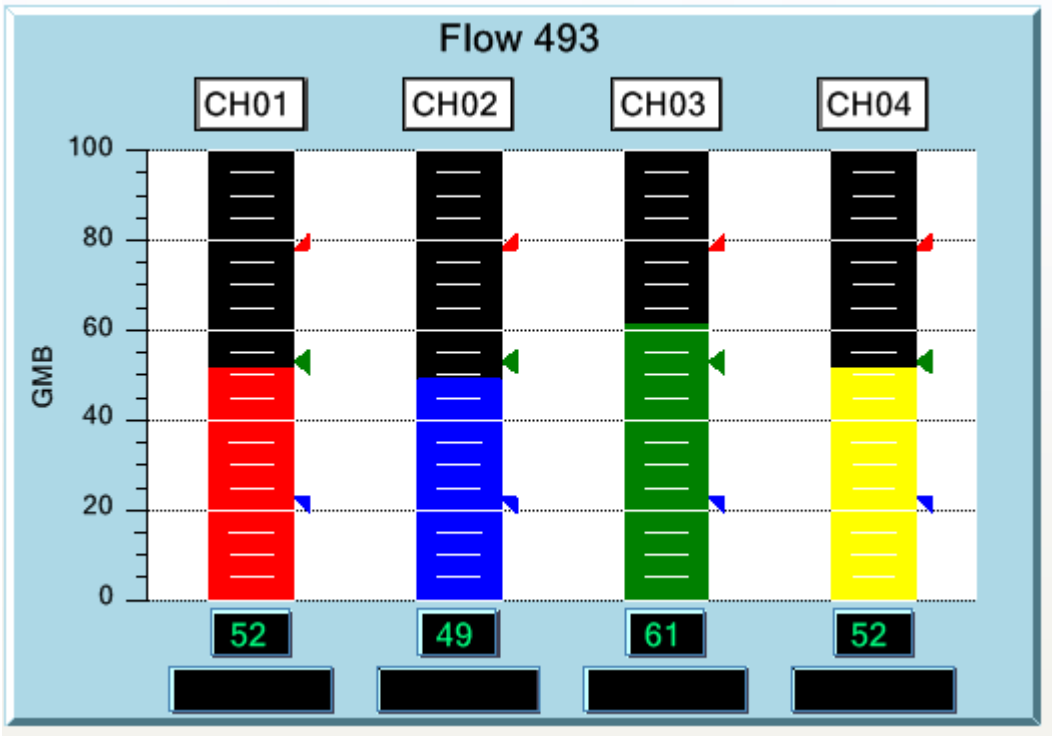
Name	Description
AlarmIndicator	Get a reference to the RTAlarmIndicator object
AlarmPanelMeter	Get a reference to the RTAlarmPanelMeter object (Inherited from RTAutoIndicator.)
BarAttributes	Set/Get the line color for the chart object.
BarDataValue	Get the numeric label template object used to place numeric values on the bars.
BarEndBulb	Set/Get to true for a bar end bulb.
BarFillColor	Set/Get the fill color for the chart object.
BarLineWidth	Set/Get the line width for the chart object.
BarOrientation	Get/Set the orientation of the chart.
BarPlot	Get a reference to the RTBarIndicator object.
BarWidth	Set/Get the bar width.
BarWidthPixels	Set/Get to the pixel width of the bar in the bar plot.
CoordinateSystem	Get the coordinate system object for the indicator. (Inherited from RTAutoIndicator.)
FaceplateBackground	Set to true to show 3D faceplate (Inherited from RTAutoIndicator.)
GraphBackground	Get the graph background object. (Inherited from RTAutoIndicator.)
GraphBorder	Get the default graph border for the chart. (Inherited from RTAutoIndicator.)
GraphFormat	Get/Set any an indicator format, is supported (Inherited from RTAutoIndicator.)
Height	Get/Set the height of the control.
HighAlarm	Get the most recent high RTAlarm object (Inherited from RTAutoIndicator.)

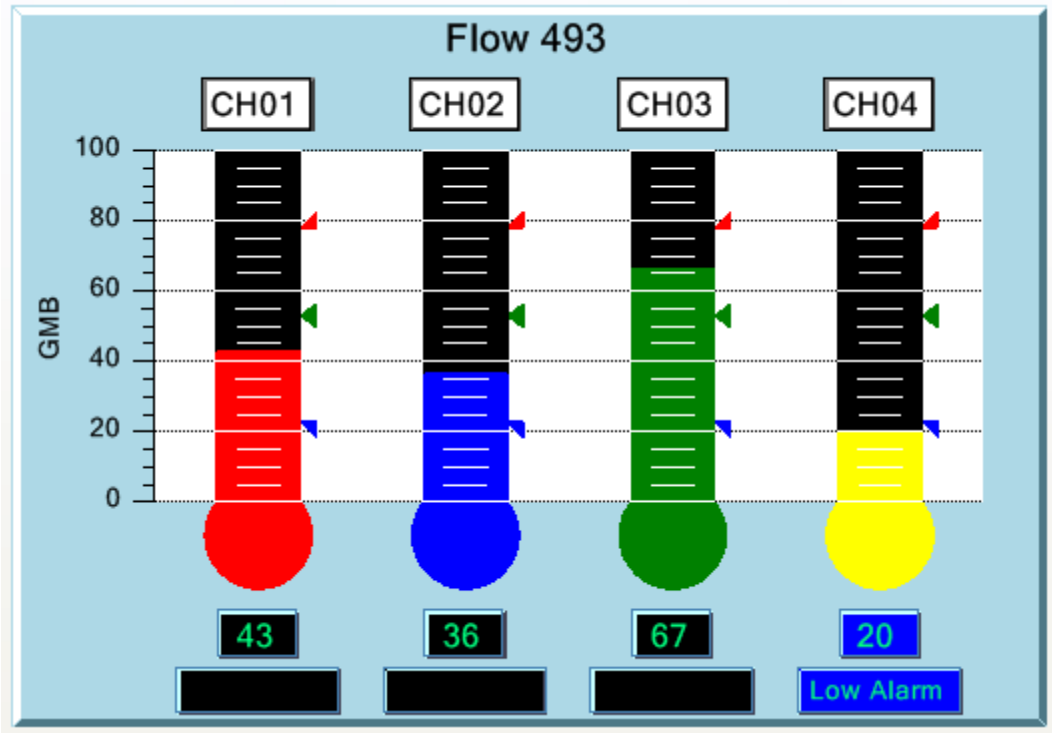
InteriorAxis	Set/Get to true and an interior axis is drawn
LowAlarm	Get the most recent low RTAlarm object (Inherited from RTAutoIndicator.)
MainTitle	Get/Set the tag string (Inherited from RTAutoIndicator.)
MaxIndicatorValue	The maximum value for the indicator. (Inherited from RTAutoIndicator.)
MinIndicatorValue	The minimum value for the indicator. (Inherited from RTAutoIndicator.)
MultiAlarmIndicator	Get a reference to the RTMultiAlarmIndicator object
MultiBarPlot .	Get a reference to the RTMultiBarIndicator object
NumericPanelMeter	Get a reference to the RTNumericPanelMeter object (Inherited from RTAutoIndicator.)
PlotAttrib	Get the ChartAttribute object associated with the plot.
PlotAttribArray	Get an array of ther attributes object for the bars of the bar graph.
PlotBackground	Get the plot background object. (Inherited from RTAutoIndicator.)
PreferredSize	Set/Get the preferred size of the control (Inherited from ChartView.) Established base size for font scaling.
ProcessVariable	Get most recently created RTProcessVar. (Inherited from RTAutoIndicator.)
RenderingMode	(Inherited from ChartView.)
ResetOnDraw	Set/Get True the ChartView object list is cleared with each redraw (Inherited from RTAutoIndicator.)
ResizeMode	(Inherited from ChartView.)
SetpointAlarm	Get the most recent setpoint RTAlarm object (Inherited from RTAutoIndicator.)
TagPanelMeter	Get a reference to the tag panel meter object (Inherited from RTAutoIndicator.)
TagString	Get/Set the tag string (Inherited from RTAutoIndicator.)
UnitsPanelMeter	Get a reference to the units string panel meter object (Inherited from RTAutoIndicator.)
UnitsString	Get/Set the units string (Inherited from RTAutoIndicator.)
Visible	Get/Set a value indicating whether the control is displayed.
Width	Get/Set the width of the control.
XAxis	Get the x-axis object.

XAxis2	Get the second x-axis object.
XAxisLab	Get the x-axis labels object.
XAxisTitle	Get the x-axis title object.
XGrid	Get the x-axis grid object.
YAxis	Get the y-axis object.
YAxis2	Get the second y-axis object.
YAxisLab	Get the y-axis labels object. Accessible only after BuildGrap
YAxisTitle	Get the y-axis title object.
YGrid	Get the y-axis grid object.

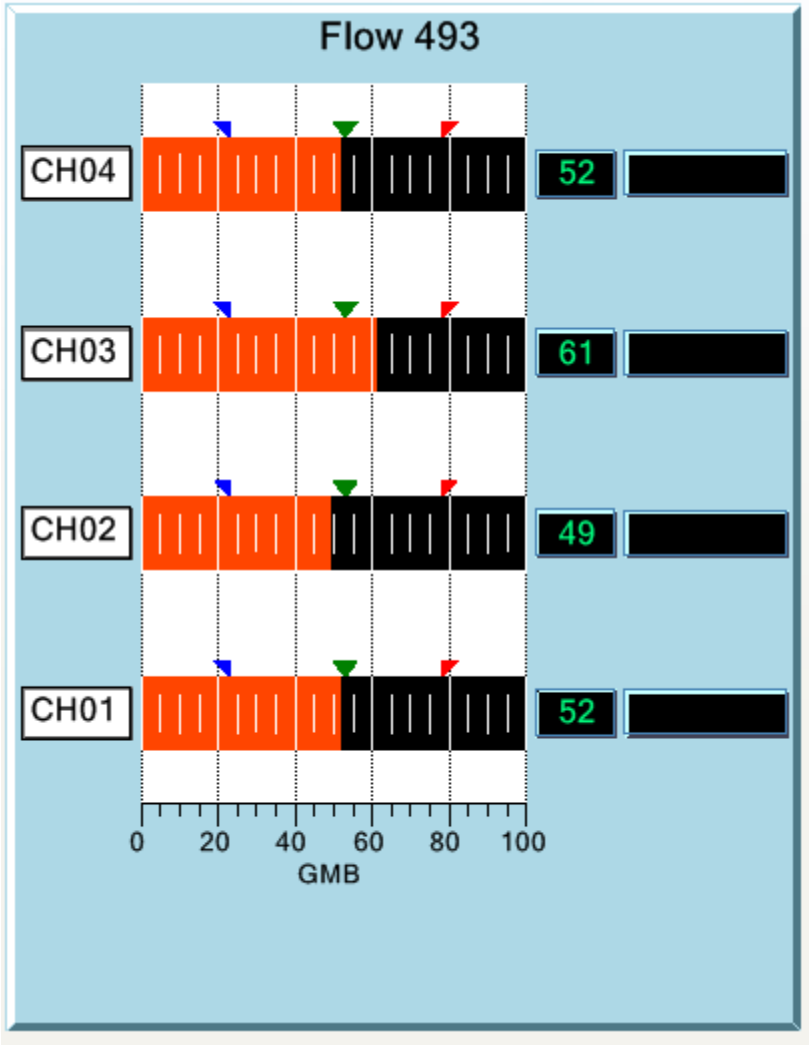
A complete listing of **RTAutoBarIndicator** properties classes is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file

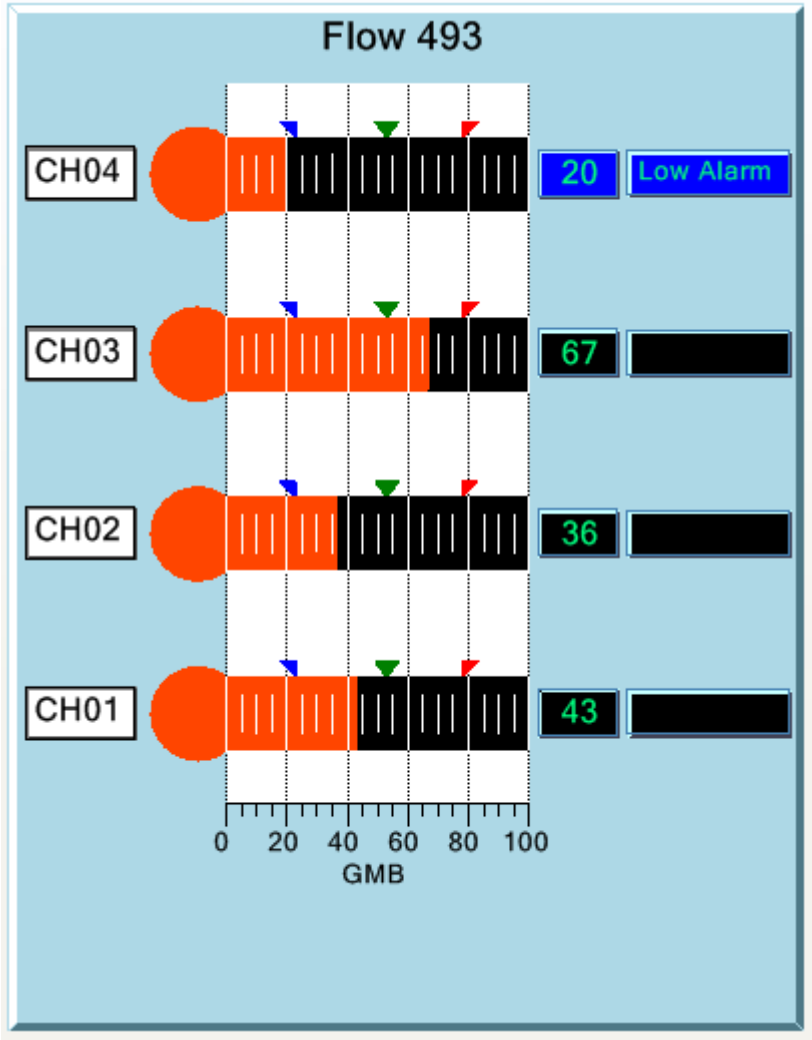
There are two different bar graph formats, horizontal and vertical. Below you will find a brief description of the differences between the formats.





Panel meters above and below the bar for the tag name, numeric value, and alarm status. The scale units displayed vertically on the left. Turn the BarEndBulb property on and the bar indicator area will rescale to fit in the bulb without overlapting the numeric and alarm status panel.

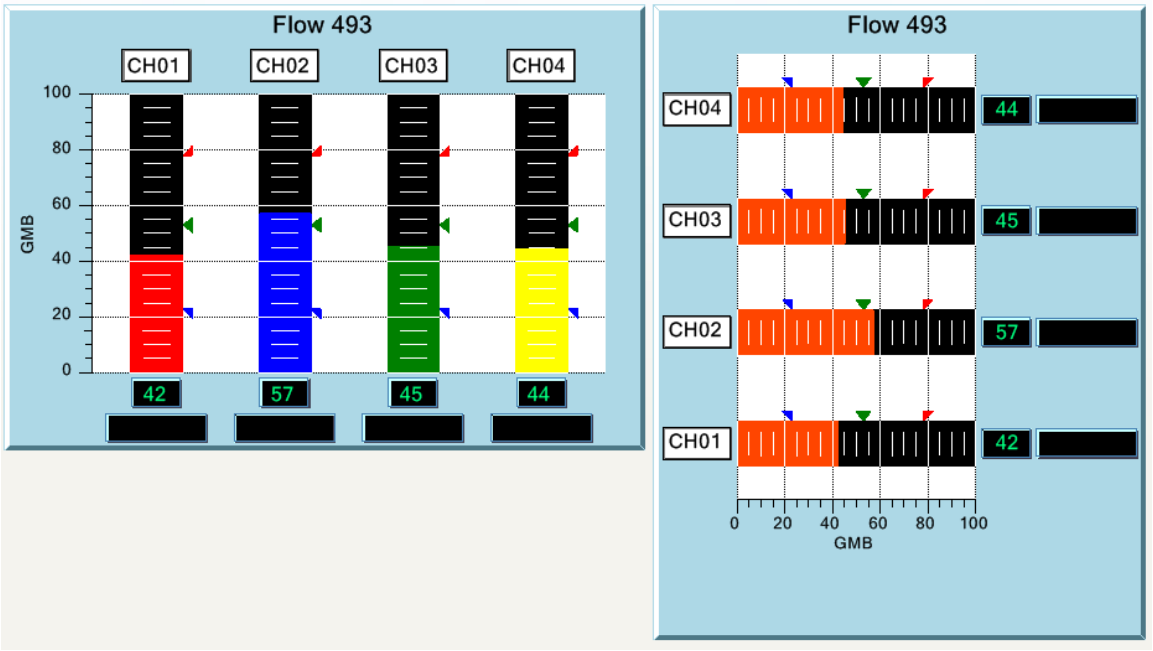

Flow 493
CH04
52
CH03
61
CH02
49
CH01
52
0
20
40
60
80
100
GMB



Panel meters to the left and right of the bar for the tag name, numeric value, and alarm status. The scale units displays horizontally under the scale. Turn the BarEndBulb property on and the bar indicator area shrinks horizontally in order to fit in the bulb without overlap.

Example for initializing RTAutoMultiBarIndicator objects

The example below, extracted from the AutoGraphDemos.AutoMultiBarIndicators example, draws four vertical and four horizontal bargraphs.



Below you will find the code used to initialize the first of the bargraphs above, extracted from the AutoGraphDemo.AutoMultiBarIndicators

```
void InitializeBargraphs(boolean barbulb, boolean interioraxis,
        boolean numeric, boolean alarm, boolean units, boolean tags,
        boolean title) {
    this.rtAutoMultiBarIndicator1.setBarOrientation(1);
    this.rtAutoMultiBarIndicator1.setLocation(3, 28);
    this.rtAutoMultiBarIndicator1.setSize(518, 360);
    this.rtAutoMultiBarIndicator1.setPreferredSize(518, 360);
    //
    // rtAutoMultiBarIndicator2
    //
    this.rtAutoMultiBarIndicator2.setBarOrientation(0);
    this.rtAutoMultiBarIndicator2.setLocation(527, 28);
    this.rtAutoMultiBarIndicator2.setSize(398, 513);
    this.rtAutoMultiBarIndicator2.setPreferredSize(398, 513);

    rtAutoMultiBarIndicator1.initMultiBarIndicator(ChartObj.VERT_DIR, 1,
            ChartColors.ORANGERED, 4);
    rtAutoMultiBarIndicator1.getMultiBarPlot().setBarWidth(0.1);
    rtAutoMultiBarIndicator1.initColors(barcolors);
    rtAutoMultiBarIndicator1.initStrings("Flow 493", "GMB", bartags);
    rtAutoMultiBarIndicator1.getLowAlarm().setAlarmLimitValue(23);
    rtAutoMultiBarIndicator1.getHighAlarm().setAlarmLimitValue(78);
    rtAutoMultiBarIndicator1.getSetpointAlarm().setAlarmLimitValue(53);

    rtAutoMultiBarIndicator1.getGraphBackground().setChartObjAttributes(
            new ChartAttribute(ChartColors.LIGHTBLUE, 5,
                    ChartConstants.LS_SOLID, ChartColors.LIGHTBLUE));
    rtAutoMultiBarIndicator1.setFaceplateBackground(true);
    rtAutoMultiBarIndicator1.setBarEndBulb(barbulb);
    rtAutoMultiBarIndicator1.setInteriorAxis(interioraxis);

    rtAutoMultiBarIndicator1.getNumericPanelMeter().setChartObjEnable(
            numeric ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
    rtAutoMultiBarIndicator1.getAlarmPanelMeter().setChartObjEnable(
            alarm ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
    rtAutoMultiBarIndicator1.getUnitsPanelMeter().setChartObjEnable(
```

```
            units ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoMultiBarIndicator1.getYAxisTitle().setChartObjEnable(
            units ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoMultiBarIndicator1.getTagPanelMeter().setChartObjEnable(
            tags ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
        rtAutoMultiBarIndicator1.getMainTitle().setChartObjEnable(
            title ? ChartObj.OBJECT_ENABLE : ChartObj.OBJECT_DISABLE);
.
.
.
```

Meter Indicator

Class RTAutoMeterIndicator

JPanel
ChartView
RTAutoIndicator
RTAutoMeterIndicator

The **RTAutoMeterIndicator** combines a **RTMeterIndicator** object with other objects needed to create a self-contained meter display. These other objects include a **RTProcessVar** variable, meter coordinates system, a meter axis and axis labels, title string, units string, alarm indicators, and panel meters used in the display of the meters numeric value, tag name, and alarm status. Since it contains a single **RTProcessVar** object, it displays a single channel of data.

RTAutoMeterIndicator constructors
Since the **RTAutoMeterIndicator** is designed to be dropped on a form, only a default constructor is used. The indicator is customized using public properties.

```
public RTAutoMeterIndicator ();
```

The initStrings.method is used to initialize the meters tag and units strings.

Method initStrings

```
public void initStrings(
    string title,
    string units
)
```

Parameters

title

The title (or tag) string.

units
The units string.

Use the updateIndicator method to update the meter indicator with new data.

Method updateIndicator

```
public void updateIndicator(
    double value,
    boolean updatedraw
)
```

Parameters

value
Update the indicator channel with this value.
updatedraw
True and the indicator is immediately updated.

Selected Public Instance Properties*

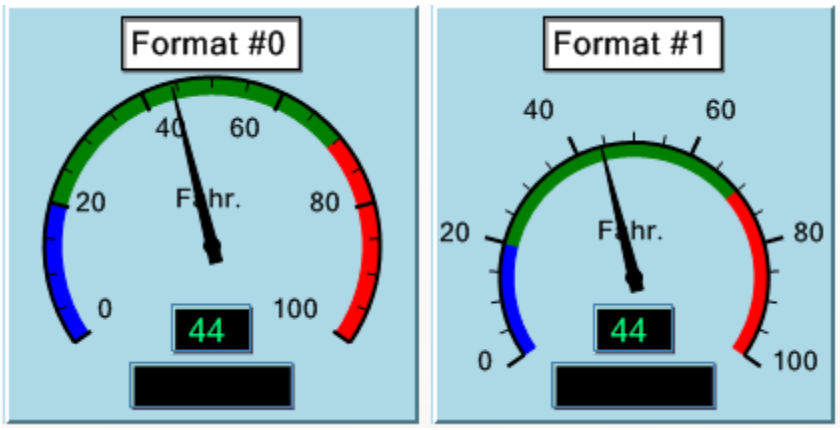
* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

Name	Description
AlarmList	Get the ArrayList holding all of the RTAlarm objects
AlarmPanelMeter	Get a reference to the RTAlarmPanelMeter object
DefaultAlarmFont	Get/Set the font used for the subhead title.
DefaultAxisLabelsFont	Get/Set the default font used for the axes labels and axes titles.
DefaultDataValueFont	Get/Set the default font used for the numeric values labeling the indicator.
DefaultFontString	Set/Get the default font used in the chart. This is a string specifiying the name of the font.
DefaultMainTitleFont	Get/Set the font used for the main title.
DefaultTagFont	Get/Set the font used for the main title.
DefaultUnitsFont	Get/Set the font used for the chart footer.
FaceplateBackground	Set to true to show 3D faceplate
GraphBackground	Get the graph background object.
GraphBorder	Get the default graph border for the chart.
GraphFormat	Get/Set any an indicator format, is supported
Height	Get/Set the height of the control.

HighAlarm	Get the most recent high RTAlarm object
LowAlarm	Get the most recent low RTAlarm object
MainTitle	Get/Set the tag string
MaxIndicatorValue	The maximum value for the indicator.
MinIndicatorValue	The minimum value for the indicator.
NumericPanelMeter	Get a reference to the RTNumericPanelMeter object
PlotAttrib	Get the ChartAttribute object associated with the plot.
PlotBackground	Get the plot background object.
PreferredSize	Set/Get the preferred size of the control (Inherited from ChartView.) Established base size for font scaling.
ProcessVariable	Get most recently created RTProcessVar.
RenderingMode	(Inherited from ChartView.)
ResizeMode	(Inherited from ChartView.)
TagPanelMeter	Get a reference to the tag panel meter object
UnitsPanelMeter	Get a reference to the units string panel meter object
UnitsString	Get/Set the units string
Visible	Get/Set a value indicating whether the control is displayed.
Width	Get/Set the width of the control.

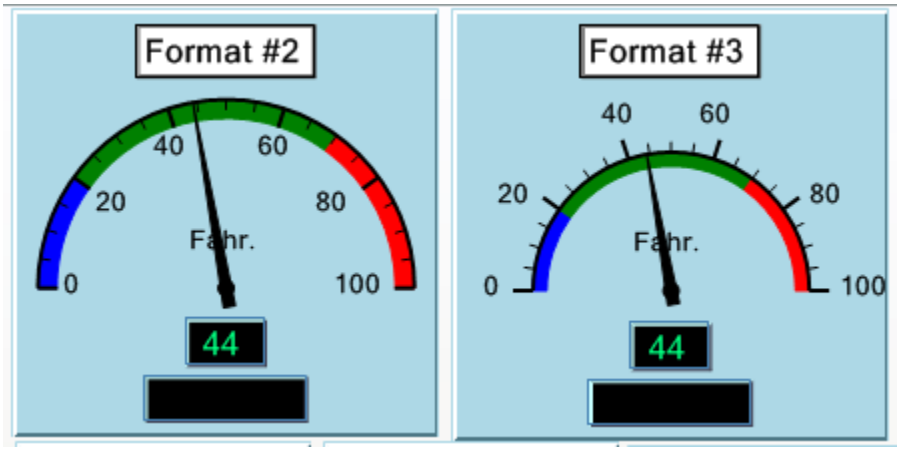
A complete listing of **RTAutoMeterIndicator** properties classes is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file

There are 12 different meter formats, four horizontal and four vertical. Use the setGraphFormat method (0..11) to set the format. Below you will find a brief description of the differences between the formats.

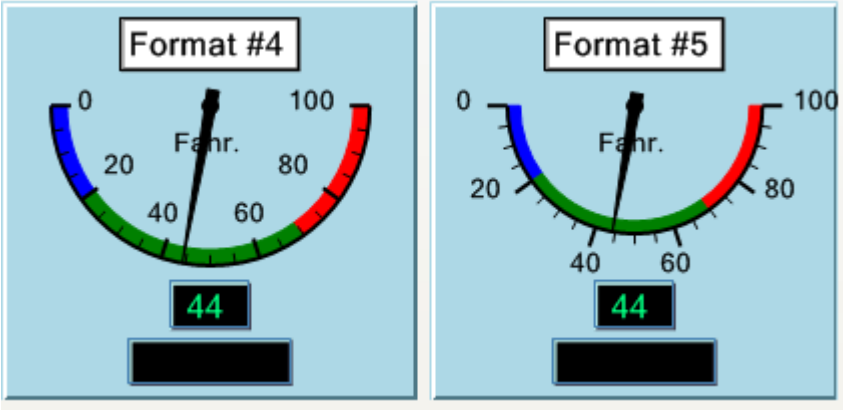


Formats #0 and #1 use 270 degree arcs (235 to -45 clockwise) with a tag string above and numeric panel meter and alarm status panel meter below the needle. The difference between the two formats is the meter ticks point inward in format #0 and outward in format #1.

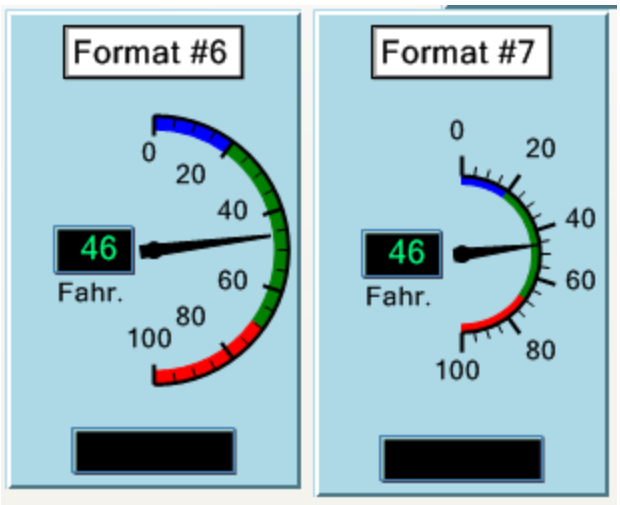
.



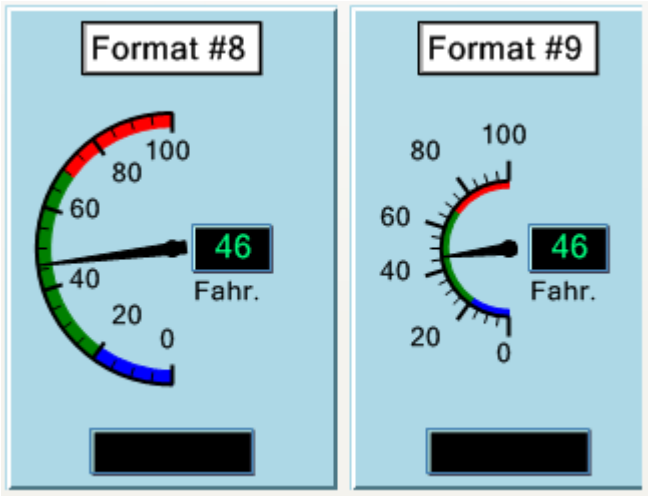
Formats #2 and #3 use 180 degree arcs (180 to 0 clockwise) with a tag string above and numeric panel meter and alarm status panel meter below the needle. The difference between the two formats is the meter ticks point inward in format #2 and outward in format #3.



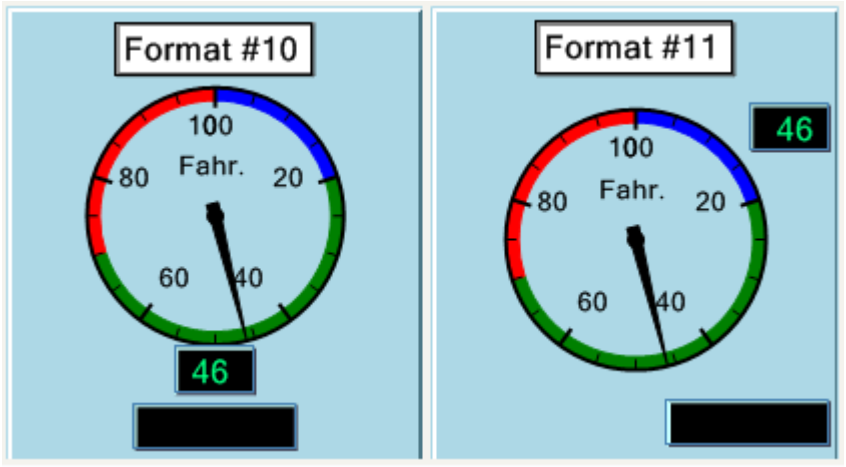
Formats #4 and #5 use 180 degree arcs (180 to 0 counter clockwise) with a tag string above and numeric panel meter and alarm status panel meter below the needle. The difference between the two formats is the meter ticks point inward in format #4 and outward in format #5.



Formats #6 and #6 use 180 degree arcs (90 to -90 clockwise) with a tag string above, numeric panel meter to to the left and alarm status panel meter below the needle. The difference between the two formats is the meter ticks point inward in format #6 and outward in format #7.



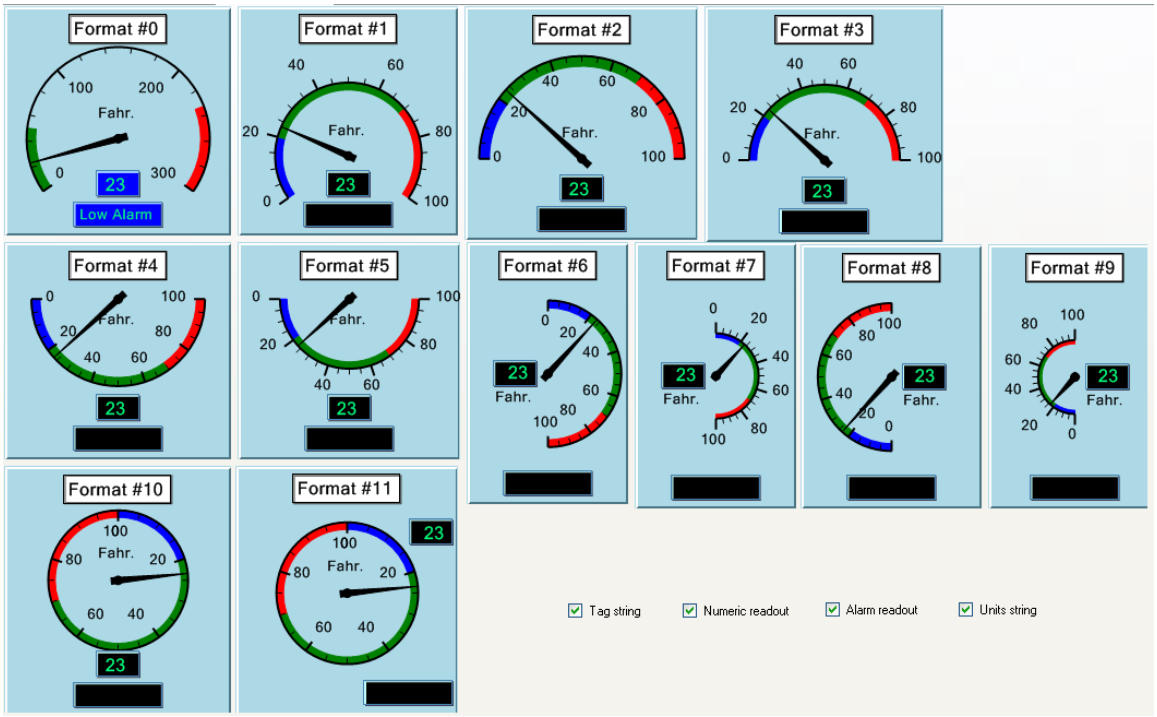
Formats #8 and #9 use 180 degree arcs (-90 to 90 clockwise) with a tag string above, numeric panel meter to to the right and alarm status panel meter below the needle. The difference between the two formats is the meter ticks point inward in format #8 and outward in format #9.



Formats #10 and #1 use 360 degree arcs 90 to 90 clockwise). Format #10 places the tag string above, and the numeric and alarm panel meters below the meter arc. Format #11 places the tag string above and the numeric and alarm panel meters to the right of the meter arc.

Example for initializing RTAutoMeterIndicator objects

The example below, extracted from the AutoGraphDemos.AutoMeterIndicators example, draws each of the 12 different meter formats.



Below you will find the code used to initialize the first of the meters above, extracted from the AutoGraphDemos.AutoNeedleMeterIndicator example program.

```
setPosition(rtAutoMeterIndicator1, new Rectangle(10,10, 175, 175));
rtAutoMeterIndicator1.setGraphFormat( 0);
rtAutoMeterIndicator1.initStrings("Format #0", "Fahr.");
rtAutoMeterIndicator1.getLowAlarm().setAlarmLimitValue(50);
rtAutoMeterIndicator1.getHighAlarm().setAlarmLimitValue(233);
rtAutoMeterIndicator1.setMinIndicatorValue ( 0);
rtAutoMeterIndicator1.setMaxIndicatorValue(300);
this.add(rtAutoMeterIndicator1);
```

Dial Indicator

Class RTAutoDialIndicator

JPanel
ChartView
RTAutoIndicator
RTAutoDialIndicator

The **RTAutoDialIndicator** combines a **RTMeterIndicator** object with other objects needed to create a self-contained meter display. These other objects include a **RTComboProcessVar** variable, meter coordinates system, a meter axis and axis labels, title string, units string, alarm indicators, and panel meters used in the display of the meters numeric value, tag name, and alarm status. Since it contains a **RTComboProcessVar** object, it can divide a single input value into multiple values to drive multiple needles in the display.

RTAutoDialIndicator constructors

Since the RTAutoDialIndicator is designed to be dropped on a form, only a default constructor is used. The indicator is customized using public properties.

```
public RTAutoDialIndicator ();
```

The initStrings.method is used to initialize the dials tag and units strings.

Method initStrings

```
public void initStrings(
    string title,
    string units
)
```

Parameters

title
: The title (or tag) string.

units
: The units string.

Use the updateIndicator method to update the dial indicator with new data.

Method updateIndicator

```
public void updateIndicator(
    double value,
    boolean updatedraw
)
```

Parameters

value
: Update the indicator channel with this value.

updatedraw
: True and the indicator is immediately updated.

Selected Public Instance Properties*

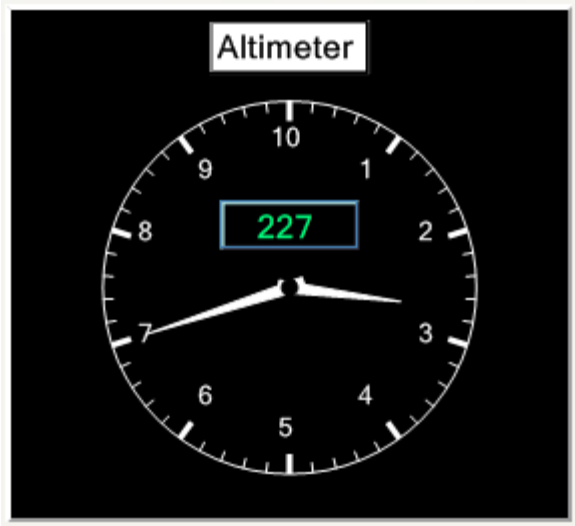
* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

Name	Description
AlarmList	Get the ArrayList holding all of the RTAlarm objects
AlarmPanelMeter	Get a reference to the RTAlarmPanelMeter object
DefaultAlarmFont	Get/Set the font used for the subhead title.
DefaultAxisLabelsFont	Get/Set the default font used for the axes labels and axes titles.
DefaultDataValueFont	Get/Set the default font used for the numeric values labeling the indicator.
DefaultFontString	Set/Get the default font used in the chart. This is a string specifiying the name of the font.
DefaultMainTitleFont	Get/Set the font used for the main title.
DefaultTagFont	Get/Set the font used for the main title.
DefaultUnitsFont	Get/Set the font used for the chart footer.
DialInterior	Get dialInterior RTGenShape object.
FaceplateBackground	Set to true to show 3D faceplate
GraphBackground	Get the graph background object.

GraphBorder	Get the default graph border for the chart.
GraphFormat	Get/Set any an indicator format, is supported
Height	Get/Set the height of the control.
HighAlarm	Get the most recent high RTAlarm object
LowAlarm	Get the most recent low RTAlarm object
MainTitle	Get/Set the tag string
MaxIndicatorValue	The maximum value for the indicator.
MinIndicatorValue	The minimum value for the indicator.
NumericPanelMeter	Get a reference to the RTNumericPanelMeter object
PlotAttrib	Get the ChartAttribute object associated with the plot.
PlotBackground	Get the plot background object.
PreferredSize	Set/Get the preferred size of the control (Inherited from ChartView.) Established base size for font scaling.
ProcessVariable	Get most recently created RTProcessVar.
RenderingMode	(Inherited from ChartView.)
ResizeMode	(Inherited from ChartView.)
TagPanelMeter	Get a reference to the tag panel meter object
UnitsPanelMeter	Get a reference to the units string panel meter object
UnitsString	Get/Set the units string
Visible	Get/Set a value indicating whether the control is displayed.
Width	Get/Set the width of the control.

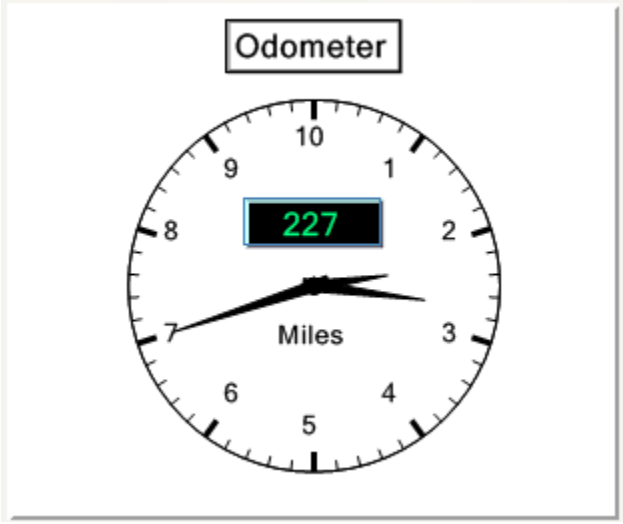
A complete listing of **RTAutoDialIndicator** properties classes is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file

There are three different dial formats. Use the setGraphFormat method (0..2) to set the format. Below you will find a brief description of the differences between the formats.

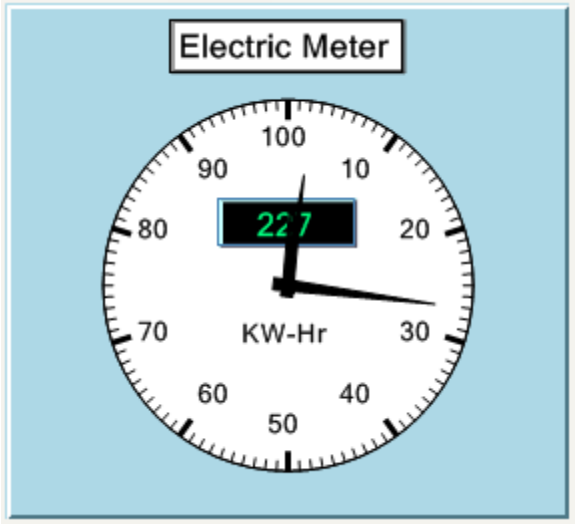


Format #0 dispalys a two needle dial, with a scale range of 0 to 10, with a tag string at the top of the windows, and a numeric panel meter above the needle pivot point.The internal RTComboProcessVar object assigns the update value to the first (longest) of the two meter needles, and the (update value) / 10 to the second (smaller) of the two needles.

.



Format #1 dispalys a three needle dial, with a scale range of 0 to 10, with a tag string at the top of the windows, and a numeric panel meter above the needle pivot point.The internal RTComboProcessVar object assigns the update value to the first (longest) of the three meter needles, the (update value) / 10 to the second (middle) of the three needles, and the (update value) / 100 to the third (shortest) of the three needles.



Format #2 dispalys a three needle dial, with a scale range of 0 to 100, with a tag string at the top of the windows, and a numeric panel meter above the needle pivot point.The internal RTComboProcessVar object assigns the update value to the first (longest) of the three meter needles, the (update value) / 100 to the second (middle) of the three needles, and the (update value) / 10000 to the third (shortest) of the three needles.

Example for initializing RTAutoDialIndicator objects

The example below, extracted from the AutoGraphDemos. AutoDialsAndClockIndicators example, draws each of the 3 different dial formats.

```
public void InitializeGraph()
{
    .
    .
    .
         this.rtAutoDialIndicator1.setGraphFormat ( 0);
         this.rtAutoDialIndicator1.getGraphBackground().setFillColor(Color.white);
         this.rtAutoDialIndicator1.invertColors();
         this.rtAutoDialIndicator1.initStrings("Altimeter", "Feet");

         this.rtAutoDialIndicator2.setGraphFormat ( 1);
         this.rtAutoDialIndicator2.getGraphBackground().setFillColor(Color.white);
         this.rtAutoDialIndicator2.invertColors();
         this.rtAutoDialIndicator2.initStrings("Odometer", "Miles");


         this.rtAutoDialIndicator3.setGraphFormat ( 2);
         this.rtAutoDialIndicator3.invertColors();
        this.rtAutoDialIndicator3.initStrings("Electric Meter", "KW-Hr");
}
```

Clock Indicator

Class RTAutoClockIndicator

ChartView
RTAutoIndicator
RTAutoClockIndicator

The **RTAutoClockIndicator** combines a **RTMeterIndicator** object with other objects needed to create a self-contained meter display. These other objects include a **RTComboProcessVar** variable, meter coordinates system, a meter axis and axis labels, title string, units string, alarm indicators, and panel meters used in the display of the meters numeric value, tag name, and alarm status. Since it contains a **RTComboProcessVar** object, it can divide a single input value (time in this case) into multiple values (hours, minutes, seconds) to drive multiple needles in the display.

RTAutoClockIndicator constructors

Since the **RTAutoClockIndicator** is designed to be dropped on a form, only a default constructor is used. The indicator is customized using public properties.

```
public RTAutoMeterIndicator ();
```

The initStrings.method is used to initialize the clock tag and units strings.

Method initStrings

```
public void initStrings(
    string title,
    string units
)
```

Parameters

title
: The title (or tag) string.

units
: The units string.

Use the UpdateClock method to update the clock indicator with a new time value.

Method updateIndicator

```
void updateClock(
    GregorianCalendar time,
    boolean updatedraw
)
```

Parameters

value

Update the clock with this time value.

updatedraw

True and the indicator is immediately updated.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

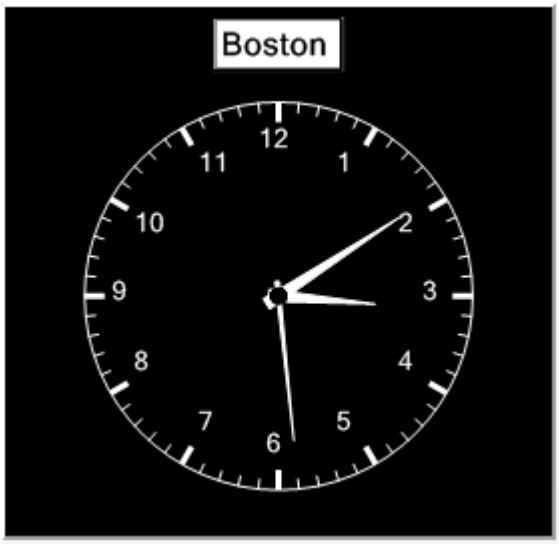
Name	Description
AlarmList	Get the ArrayList holding all of the RTAlarm objects
AlarmPanelMeter	Get a reference to the RTAlarmPanelMeter object
DefaultAlarmFont	Get/Set the font used for the subhead title.
DefaultAxisLabelsFont	Get/Set the default font used for the axes labels and axes titles.
DefaultDataValueFont	Get/Set the default font used for the numeric values labeling the indicator.
DefaultFontString	Set/Get the default font used in the chart. This is a string specifiying the name of the font.
DefaultMainTitleFont	Get/Set the font used for the main title.
DefaultTagFont	Get/Set the font used for the main title.
DefaultUnitsFont	Get/Set the font used for the chart footer.
DialInterior	Get dialInterior RTGenShape object.
FaceplateBackground	Set to true to show 3D faceplate
GraphBackground	Get the graph background object.
GraphBorder	Get the default graph border for the chart.
GraphFormat	Get/Set any an indicator format, is supported
Height	Get/Set the height of the control.
HighAlarm	Get the most recent high RTAlarm object
LowAlarm	Get the most recent low RTAlarm object
MainTitle	Get/Set the tag string

MaxIndicatorValue	The maximum value for the indicator.
MinIndicatorValue	The minimum value for the indicator.
NumericPanelMeter	Get a reference to the RTNumericPanelMeter object
PlotAttrib	Get the ChartAttribute object associated with the plot.
PlotBackground	Get the plot background object.
PreferredSize	Set/Get the preferred size of the control (Inherited from ChartView.) Established base size for font scaling.
ProcessVariable	Get most recently created RTProcessVar.
RenderingMode	(Inherited from ChartView.)
ResizeMode	(Inherited from ChartView.)
TagPanelMeter	Get a reference to the tag panel meter object
UnitsPanelMeter	Get a reference to the units string panel meter object
UnitsString	Get/Set the units string
Visible	Get/Set a value indicating whether the control is displayed.
Width	Get/Set the width of the control.

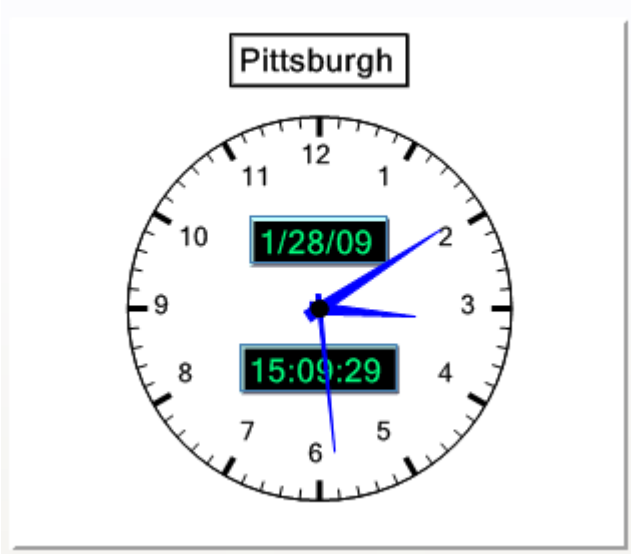
A complete listing of **RTAutoClockIndicator** properties classes is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file

.

There are two different clock formats. Use the setGraphFormat method (0..1) to set the format. Below you will find a brief description of the differences between the formats.



Format #0 displays a three hand (hours, minutes, seconds) clock. A tag name is displayed above the clock face.
.



Format #1 displays a three hand (hours, minutes, seconds) clock. The date is displayed above the center of the clock, and a digial readout of the time below the center of the clock. A tag name is displayed above the clock face.

Example for initializing RTAutoClockIndicator objects

The example below, extracted from the AutoGraphDemos. AutoDialsAndClockIndicators example, the different clock formats.

```
public void InitializeGraph()
{
    .
    .
    .
        this.rtAutoClockIndicator1.setGraphFormat ( 0);
        this.rtAutoClockIndicator1.getGraphBackground().setFillColor(Color.white);
        this.rtAutoClockIndicator1.invertColors();
        this.rtAutoClockIndicator1.initStrings("Boston", "EST");

        this.rtAutoClockIndicator2.setGraphFormat ( 1);
        this.rtAutoClockIndicator2.getGraphBackground().setFillColor(Color.white);
        this.rtAutoClockIndicator2.getPlotAttrib().setFillColor( Color.blue);
        this.rtAutoClockIndicator2.initStrings("Pittsburgh", "EST");    .
    .

}
```

Scrolling Graph (Horizontal) Indicator

Class RTAutoScrollGraph

ChartView
RTAutoIndicator
RTAutoBarIndicator
RTAutoScrollGraph

The **RTAutoScrollGraph** is a **ChartView** derived object that encapsulates all of the chart elements needed to draw a horizontal scrolling graph, including an array of **RTProcessVar** objects, a coordinate system, axes, axes labels, **RTSingleValuePlot**, **RTGroupMultiValuePlot**, **RTAlarmSymbol** alarm symbols, legend, title, subhead, footer and units strings. The **RTAutoScrollGraph** support horizontal scrolling only. Use the **RTAutoVerticalScrollGraph** for a vertical scrolling version.

There are three types of scrolling x-scales you can use in a scrolling chart: a date/time scale, an elapsed time scale or a numeric scale. Use the appropriate InitRTAutoScroll overload to select which one is most applicable to your data.

RTAutoScrollGraph constructors

Since the **RTAutoScrollGraph** is designed to be dropped on a form, only a default constructor is used. The indicator is customized using public properties.

```
public RTAutoScrollGraph ();
```

A couple of methods are used to initialize the scroll graph after instantiation, initRTAutoScrollGraph and initStrings.

The initRTAutoScrollGraph method initializes the the x- and y-scales of the scrolling graph

Method initRTAutoScrollGraph

Initialize x-scale to a Date/Time scale using **ChartCalendar** objects, linear y-scale

```
public void initRTAutoScrollGraph(
    GregorianCalendar minx,
    double miny,
    GregorianCalendar maxx,
    double maxy
)
```

Initialize x-scale to a linear scale using doubles, linear y-scale

```
public void initRTAutoScrollGraph(
    double minx,
    double miny,
    double maxx,
    double maxy
)
```

Initialize x-scale to an elapsed time scale using **ChartTimeSpan** objects, linear y-scale

```
public void initRTAutoScrollGraph(
    ChartTimeSpan minx,
    double miny,
    ChartTimeSpan maxx,
    double maxy
)
```

Initialize x-scale to the scale type specified by the parameter scaltype, linear y-scale. Use millisecond values for minx and maxx.

```
public void initRTAutoScrollGraph(
    double minx,
    double miny,
    double maxx,
    double maxy,
    int scaletype
)
```

Parameters

minx

The starting x-value as a **GregorianCalendar**, double or **ChartTimeSpan** value, depending on which of the overloads is used.

miny

The starting y-value.

maxx

The ending x-value as a **GregorianCalendar** , double or **ChartTimeSpan** value, depending on which of the overloads is used.

maxy

The ending y-value.

The initStrings method initialized the title and units strings.

Method initStrings

```
public void initStrings(
    string title,
    string units
)
```

Parameters

title

The title string.

units
The units string.

Add a channel (a plot object) to the scrolling graph using the AddRTPlotObject.

Method addRTPlotObject

```
public void addRTPlotObject(
    int plottype,
    Color colr,
    string tag
)
```

Parameters

plottype
Specifies the simple plot type: LINE_MARKER_PLOT, LINE_PLOT, BAR_PLOT, SCATTER_PLOT

colr
The primary color of the plot object.

tag
The tag name associated with the plot object.

Use the **updateIndicator method to update the scrolling graph with new data.**

Method updateIndicator

```
public void updateIndicator(
    double[] values,
    boolean updatedraw
)
public void updateIndicator(
    double value,
    boolean updatedraw
)
```

Parameters

values
An array of new values, one for each channel of the indicator.

value
A single value if the scroll graphs only has one channel..

updatedraw
True and the indicator is immediately updated.

Selected Public Instance Properties*

* Java does not have properties but we use that terminology never less - programmers add set or get in front of property names, i.e. **setAspectRatioCorrection**(value) and value := **getAspectRatioCorrection** ();

Name	Description
AlarmList	Get the ArrayList holding all of the RTAlarm objects (Inherited from RTAutoIndicator.)
AlarmPanelMeter	Get a reference to the RTAlarmPanelMeter object (Inherited from RTAutoIndicator.)
BarDataValue	Get the numeric label template object used to place numeric values on the bars.
BarFillColor	Sets the fill color for the chart object.
BarLineColor	Sets the line color for the chart object.
BarLineWidth	Sets the line width for the chart object.
BarWidth	Set/Get the bar width.
ChartLegend	Get/Set the charts Legend object
ChartObjType	Get/Set the chart object type. (Inherited from RTAutoIndicator.)
ChartSimpleDataset	Get the SimpleDataset object that holds the data used to plot the scroll graph.
CoordinateSystem	Get the coordinate system object for the indicator. (Inherited from RTAutoIndicator.)
DatasetList	Get dataset object list.
Datatooltip	Get the data tooltip object for the chart.
DefaultChartFontString	Set/Get the default font used in the chart. This is a string specifiying the name of the font.
DefaultLegendFont	Get/Set the font used for the legend.
DefaultSubHeadFont	Get/Set the font used for the sub head.
DefaultToolTipFont	Set/Get the default font object used for the tooltip.
DrawEnable	(Inherited from ChartView.)
FaceplateBackground	Set to true to show 3D faceplate (Inherited from RTAutoIndicator.)
GraphBackground	Get the graph background object. (Inherited from RTAutoIndicator.)
GraphBorder	Get the default graph border for the chart. (Inherited from RTAutoIndicator.)
GraphFormat	Get/Set any an indicator format, is supported (Inherited from RTAutoIndicator.)
GraphScrollFrame	Get the graphs RTScrollFrame.
GroupPlotObj	Get the GroupVersaPlot plot object.
Height	Get/Set the height of the control.

HighAlarm	Get the most recent high RTAlarm object (Inherited from RTAutoIndicator.)
LowAlarm	Get the most recent low RTAlarm object (Inherited from RTAutoIndicator.)
MainTitle	Get/Set the tag string (Inherited from RTAutoIndicator.)
MaxIndicatorValue	The maximum value for the indicator. (Inherited from RTAutoIndicator.)
MinIndicatorValue	The minimum value for the indicator. (Inherited from RTAutoIndicator.)
NumericPanelMeter	Get a reference to the RTNumericPanelMeter object (Inherited from RTAutoIndicator.)
PlotAttrib	Get the ChartAttribute object associated with the plot.
PlotBackground	Get the plot background object. (Inherited from RTAutoIndicator.)
PlotObjectList	Get plot object list.
PreferredSize	Set/Get the preferred size of the control (Inherited from ChartView.) Established base size for font scaling.
ProcessVariable	Get most recently created RTProcessVar. (Inherited from RTAutoIndicator.)
ProcessVariableArray	Get the ArrayList holding all of the RTProcessVar objects
ResetOnDraw	Set/Get True the ChartView object list is cleared with each redraw (Inherited from RTAutoIndicator.)
ResizeMode	(Inherited from ChartView.)
SetpointAlarm	Get the most recent setpoint RTAlarm object (Inherited from RTAutoIndicator.)
SimplePlotObj	Get the SimpleVersaPlot plot object.
SingleValuePlot	Get the most recent RTSingleValuePlot object
SingleValuePlotList	Get the ArrayList holding all of the RTSimpleSingleValuePlot objects
SmoothingMode	(Inherited from ChartView.)
SubHead	Get the sub head object for the chart.
TagPanelMeter	Get a reference to the tag panel meter object (Inherited from RTAutoIndicator.)
TagString	Get/Set the tag string (Inherited from RTAutoIndicator.)

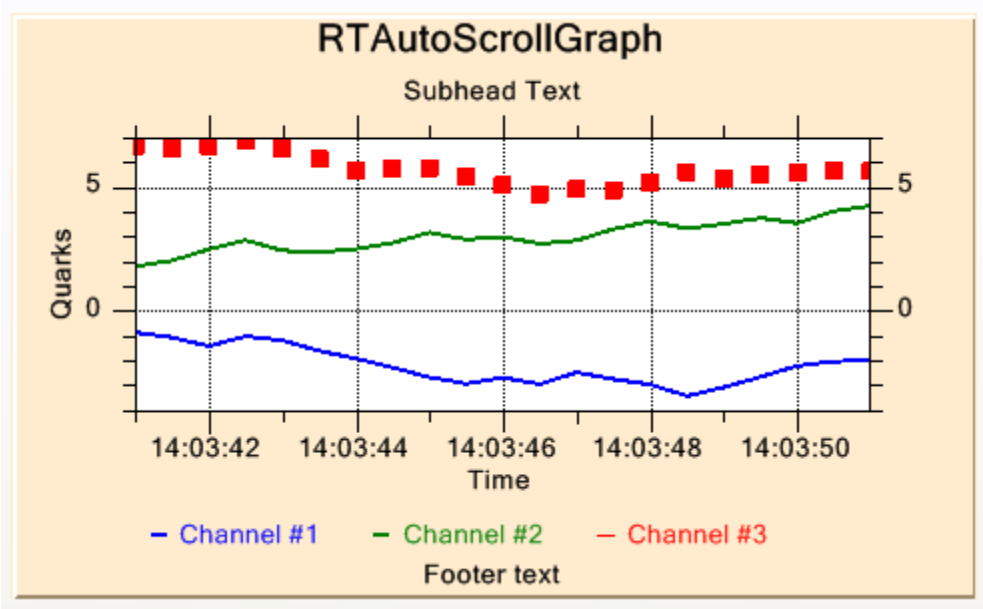
TextRenderingHint	(Inherited from ChartView.)
UnitsPanelMeter	Get a reference to the units string panel meter object (Inherited from RTAutoIndicator.)
UnitsString	Get/Set the units string (Inherited from RTAutoIndicator.)
Visible	Get/Set a value indicating whether the control is displayed.
Width	Get/Set the width of the control.
XAxis	Get the x-axis object.
XAxis2	Get the second x-axis object.
XAxisLab	Get the x-axis labels object.
XAxisLab2	Get the second x-axis labels object.
XAxisTitle	Get the x-axis title object.
XGrid	Get the x-axis grid object.
YAxis	Get the y-axis object.
YAxis2	Get the second y-axis object.
YAxisLab	Get the y-axis labels object. Accessible only after BuildGrap
YAxisLab2	Get the second y-axis labels object. Accessible only after BuildGrap
YAxisTitle	Get the y-axis title object.
YGrid	Get the y-axis grid object.

A complete listing of **RTAutoScrollGraph** properties classes is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file.

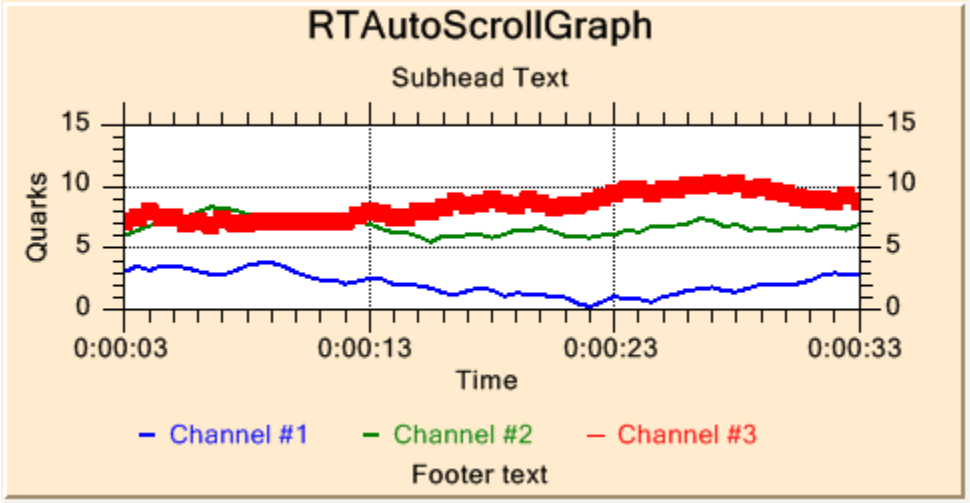
The **RTAutoScrollGraph** class supports a date/time, elapsed time and numeric time horizontal (x-axis) scale. The y-axis scale is linear or logarithmic.The title and subhead shows above above the chart, the legend and footer below.

Example for initializing RTAutoScrollGraph objects

The example below, extracted from the AutoGraphDemos. SimpleAutoScrollUserControl1 example, draws horizontal and vertical scrolling graphs. The top horizontal scrolling graph uses a date/time scale for the scrolling axis. The bottom horizontal scroll graph uses an elapsed time scale for the scrolling axis.



Scrolling graph with Date/Time x-axis scale



Scrolling graph with elapsed time x-axis scale

```
public void InitializeGraph()
{

  GregorianCalendar starttime = new GregorianCalendar();
  GregorianCalendar endtime = new GregorianCalendar();
  endtime.add(GregorianCalendar.SECOND,10);

  rtAutoScrollGraph1.initRTAutoScrollGraph(starttime,0, endtime,15);
  rtAutoScrollGraph1.initSimpleRTPlotObject(ChartObj.LINEPLOT, Color.blue,
"Channel #1");
```

```
  rtAutoScrollGraph1.initSimpleRTPlotObject(ChartObj.LINEPLOT, Color.green,
"Channel #2");
  rtAutoScrollGraph1.initSimpleRTPlotObject(ChartObj.SCATTERPLOT, Color.red,
"Channel #3");
  rtAutoScrollGraph1.getSimplePlotObj().getSymbolAttributes().setSymbolSize( 8);
  rtAutoScrollGraph1.getGraphScrollFrame().setScrollRescaleMargin (0.01);

rtAutoScrollGraph1.getGraphScrollFrame().setScrollScaleModeY( ChartObj.RT_AUTOSCAL
E_Y_MINMAX);
  rtAutoScrollGraph1.initStrings("RTAutoScrollGraph", "Time", "Quarks");
  rtAutoScrollGraph1.getGraphBackground().setFillColor
(ChartColors.BLANCHEDALMOND);
  rtAutoScrollGraph1.getSubHead().setTextString ( "Time/Date Scrolling");
  rtAutoScrollGraph1.getSubHead().setChartObjEnable(ChartObj.OBJECT_ENABLE);
  rtAutoScrollGraph1.getFooter().setTextString("Footer text");
  rtAutoScrollGraph1.getFooter().setChartObjEnable(ChartObj.OBJECT_ENABLE);

rtAutoScrollGraph1.getYAxisLab2().setChartObjEnable(ChartObj.OBJECT_ENABLE);
rtAutoScrollGraph1.getLowAlarm().setAlarmLimitValue( 4);
rtAutoScrollGraph1.getHighAlarm().setAlarmLimitValue(12);
rtAutoScrollGraph1.getSetpointAlarm().setAlarmLimitValue(8);

rtAutoScrollGraph1.getChartLegend().setChartObjEnable(ChartConstants.OBJECT_DISABL
E);



.
.
.


ChartTimeSpan startts = ChartTimeSpan.fromSeconds(0);
ChartTimeSpan endts = ChartTimeSpan.fromSeconds(30);
rtAutoScrollGraph2.initRTAutoScrollGraph(startts, 0, endts, 15);
rtAutoScrollGraph2.initSimpleRTPlotObject(ChartObj.LINEPLOT, Color.blue, "Channel
#1");
rtAutoScrollGraph2.initSimpleRTPlotObject(ChartObj.LINEPLOT, Color.green, "Channel
#2");
rtAutoScrollGraph2.initSimpleRTPlotObject(ChartObj.SCATTERPLOT, Color.red,
"Channel #3");
rtAutoScrollGraph2.getSimplePlotObj().getSymbolAttributes().setSymbolSize( 8);
rtAutoScrollGraph2.getGraphScrollFrame().setScrollRescaleMargin (0.01);
rtAutoScrollGraph2.getGraphScrollFrame().setScrollScaleModeY( ChartObj.RT_AUTOSCAL
E_Y_MINMAX);
rtAutoScrollGraph2.initStrings("RTAutoScrollGraph", "Time", "Quarks");
rtAutoScrollGraph2.getGraphBackground().setFillColor (ChartColors.BLANCHEDALMOND);
rtAutoScrollGraph2.getSubHead().setTextString ( "Time/Date Scrolling");
rtAutoScrollGraph2.getSubHead().setChartObjEnable(ChartObj.OBJECT_ENABLE);
rtAutoScrollGraph2.getFooter().setTextString("Footer text");
rtAutoScrollGraph2.getFooter().setChartObjEnable(ChartObj.OBJECT_ENABLE);

rtAutoScrollGraph2.getYAxisLab2().setChartObjEnable(ChartObj.OBJECT_ENABLE);
rtAutoScrollGraph2.getLowAlarm().setAlarmLimitValue( 4);
rtAutoScrollGraph2.getHighAlarm().setAlarmLimitValue(12);
rtAutoScrollGraph2.getSetpointAlarm().setAlarmLimitValue(8);

startCalendar = new GregorianCalendar();

eventTimer1.start();

}
```

The update of scroll graphs takes place in the timer event handler.

```
private void timer1_Tick(ActionEvent event)
{

    GregorianCalendar timestamp = new GregorianCalendar();
```

```
    for (int i = 0; i < currentValues.length; i++)
       currentValues[i] += (0.5 - ChartSupport.getRandomDouble());
    rtAutoScrollGraph1.updateScrollGraph(timestamp, currentValues, true);


    // get elapsed time in milliseconds
   double etimemsecs = timestamp.getTimeInMillis() -
startCalendar.getTimeInMillis();
   ChartTimeSpan etimespan = ChartTimeSpan.fromMilliseconds(etimemsecs);
   rtAutoScrollGraph2.updateScrollGraph(etimespan, currentValues, true);

   count++;
.
.
.

}
```

Scrolling Graph (Vertical) Indicator

Class RTAutoVerticalScrollGraph

JPanel
ChartView
RTAutoIndicator
RTAutoBarIndicator
RTAutoVerticalScrollGraph

The **RTAutoVerticalScrollGraph** is a **ChartView** derived object that encapsulates all of the chart elements needed to draw a scrolling graph, including an array of **RTProcessVar** objects, a coordinate system, axes, axes labels, **RTSingleValuePlot**, **RTGroupMultiValuePlot**, **RTAlarmSymbol** alarm symbols, legend, title, subhead, footer and units strings. The **RTAutoVerticalScrollGraph** support vertical horizontal scrolling. Use the **RTAutoScrollGraph** for horizontal scrolling.

There are three types of scrolling y-scales you can use in a scrolling chart: a date/time scale, an elapsed time scale or a numeric scale. Use the appropriate InitRTAutoScroll overload to select which one is most applicable to your data.

RTAutoVerticalScrollGraph constructors

Since the **RTAutoVerticalScrollGraph** is designed to be dropped on a form, only a default constructor is used. The indicator is customized using public properties.

```
public RTAutoVerticalScrollGraph ();
```

A couple of methods are used to initialize the scroll graph after instantiation, InitRTAutoScrollGraph and initStrings.

The initRTAutoScrollGraph method initializes the the x- and y-scales of the scrolling graph. Note that the y-values are now the time-based scale, rather than the x-values as in the RTAutoScrollGraph class.
Initialize the x-scale a linear scale and the y-scale to a Date/Time scale using ChartCalendar objects.

```
public void initRTAutoScrollGraph(
    double minx,
    GregorianCalendar miny,
    double maxx
    GregorianCalendar maxy,
)
```

Initialize the x-scale to a linear scale and the y-scale to a linear scale

```
public void initRTAutoScrollGraph(
    double minx,
    double miny,
    double maxx,
    double maxy
)
```

Initialize the x-scale to a linear scale and the y-scale to an elapsed time using ChartTimeSpan objects

```
public void initRTAutoScrollGraph(
    double minx,
    ChartTimeSpan miny,
    double maxx
    ChartTimeSpan maxy,
)
```

Initialize x-scale to a linear scale and the y-scale to the scale type specified by the parameter. Use millisecond values for miny and maxy.

```
public void initRTAutoScrollGraph(
    double minx,
    double miny,
    double maxx,
    double maxy,
    int scaletype
)
```

Parameters

minx
: The starting x-value.

miny
: The starting y-value as a GregorianCalendar , double or ChartTimeSpan value, depending on which of the overloads is used.

maxx
: The ending x-value.

maxy

The ending y-value as a GregorianCalendar , double or ChartTimeSpan value, , depending on which of the overloads is used.

The initStrings method initialized the title and units strings.

Method initStrings

```
public void initStrings(
    string title,
    string units
)
```

Parameters

title

The title string.

units

The units string.

Add a channel (a plot object) to the scrolling graph using the AddRTPlotObject.

Method addRTPlotObject

```
public void AddRTPlotObject(
    int plottype,
    Color colr,
    string tag
)
```

Parameters

plottype

Specifies the simple plot type: LINE_MARKER_PLOT, LINE_PLOT, BAR_PLOT, SCATTER_PLOT

colr

The primary color of the plot object.

tag

The tag name associated with the plot object.

Use the **updateIndicator method to update the scrolling graph with new data.**

Method updateIndicator

```
public void updateIndicator(
    double[] values,
    boolean updatedraw
)
public void updateIndicator(
    double value,
```

```
    boolean updatedraw
)
```

Parameters

values
: An array of new values, one for each channel of the indicator.

value
: A single value if the scroll graphs only has one channel..

updatedraw
: True and the indicator is immediately updated.

Selected Public Instance Properties

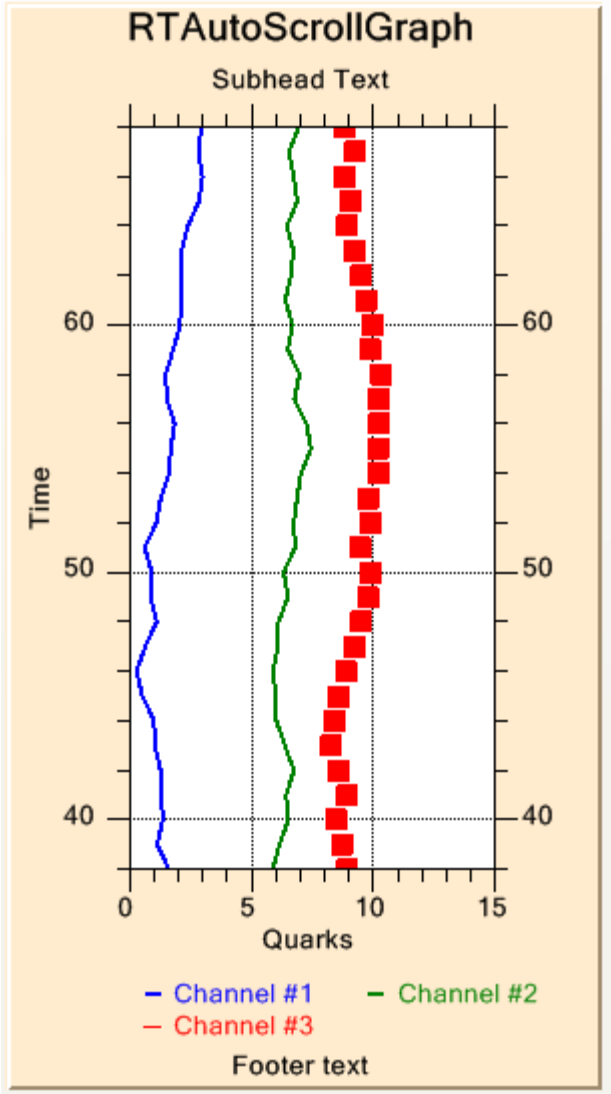
Refer to the list of properties under the RTAutoScrollGraph description.

A complete listing of **RTAutoVerticalScrollGraph** properties is found in the Javadoc documentation found in the \doc folder of the QCRTGraphJava.jar library file..

The **RTAutoVerticalScrollGraph** class supports a date/time, elapsed time and numeric time horizontal (x-axis) scale. The y-axis scale is linear or logarithmic.The title and subhead shows above above the chart, the legend and footer below.

Example for initializing RTAutoVerticalScrollGraph objects

The example below, extracted from the AutoGraphDemos. SimpleAutoScrollUserControl1 example, draws horizontal and vertical scrolling graphs. The rightmost scrolling graph is a vertical scrolling that uses a linear (numeric) scale.



Vertical scrolling graph numeric y-scale

```
public void InitializeGraph()
{
    .
    .
    .
        double startvalue = 0;
        double stopvalue = 30;
        rtAutoVerticalScrollGraph1.initRTAutoScrollGraph(0, startvalue, 15, stopvalue);
        rtAutoVerticalScrollGraph1.initSimpleRTPlotObject(ChartObj.LINEPLOT, Color.blue,
        "Channel #1");
        rtAutoVerticalScrollGraph1.initSimpleRTPlotObject(ChartObj.LINEPLOT, Color.green,
        "Channel #2");
        rtAutoVerticalScrollGraph1.initSimpleRTPlotObject(ChartObj.SCATTERPLOT, Color.red,
        "Channel #3");


        rtAutoVerticalScrollGraph1.getSimplePlotObj().getSymbolAttributes().setSymbolSize(
        8);
        rtAutoVerticalScrollGraph1.getGraphScrollFrame().setScrollRescaleMargin (0.01);
```

```
        rtAutoVerticalScrollGraph1.getGraphScrollFrame().setScrollScaleModeY( ChartObj.RT_
        AUTOSCALE_Y_MINMAX);
               rtAutoVerticalScrollGraph1.initStrings("RTAutoScrollGraph", "Time",
        "Quarks");
               rtAutoVerticalScrollGraph1.getGraphBackground().setFillColor
        (ChartColors.BLANCHEDALMOND);
               rtAutoVerticalScrollGraph1.getSubHead().setTextString ( "Time/Date
        Scrolling");

        rtAutoVerticalScrollGraph1.getSubHead().setChartObjEnable(ChartObj.OBJECT_ENABLE);
        rtAutoVerticalScrollGraph1.getFooter().setTextString("Footer text");

        rtAutoVerticalScrollGraph1.getFooter().setChartObjEnable(ChartObj.OBJECT_ENABLE);

        rtAutoVerticalScrollGraph1.getYAxisLab2().setChartObjEnable(ChartObj.OBJECT_ENABLE
        );
        rtAutoVerticalScrollGraph1.getLowAlarm().setAlarmLimitValue( 4);
        rtAutoVerticalScrollGraph1.getHighAlarm().setAlarmLimitValue(12);
        rtAutoVerticalScrollGraph1.getSetpointAlarm().setAlarmLimitValue(8);

    .
    .
    .

}
```

The update of scroll graphs takes place in the timer event handler.

```
        private void timer1_Tick(ActionEvent event)
        {

            for (int i = 0; i < currentValues.length; i++)
               currentValues[i] += (0.5 - ChartSupport.getRandomDouble());
        .
        .
        .


           count++;
           rtAutoVerticalScrollGraph1.updateScrollGraph(count, currentValues, true);
        }
```

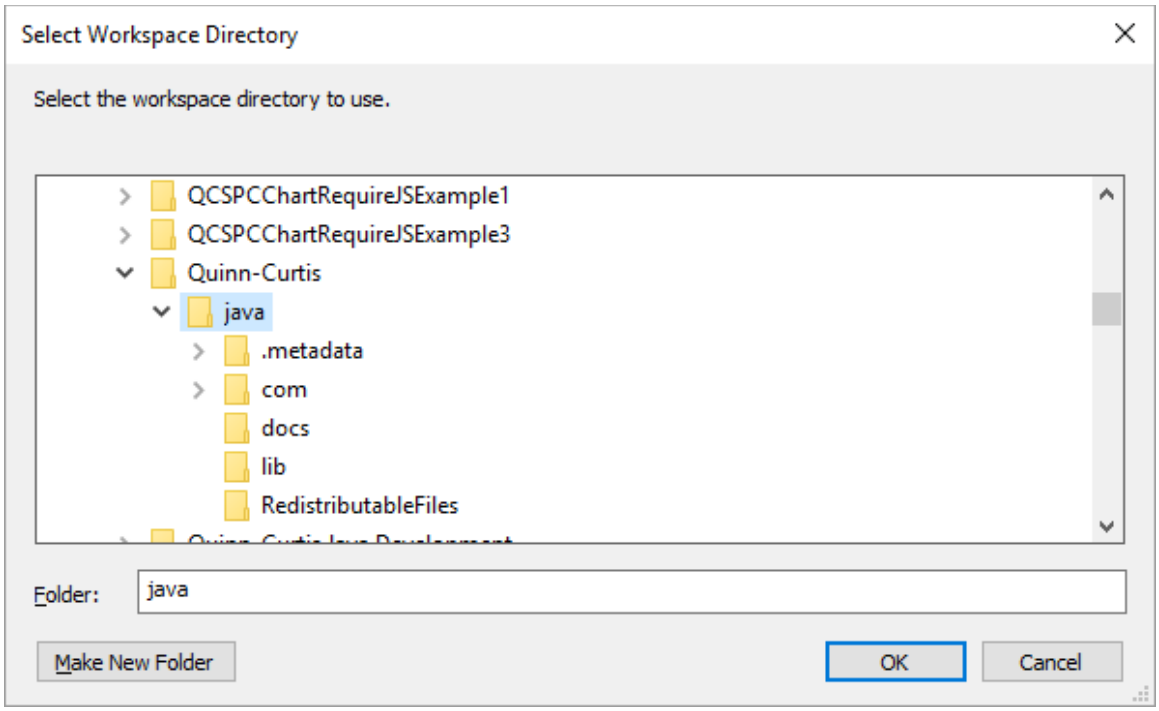
20. Compiling and Running the Example Programs

Using Eclipse IDE

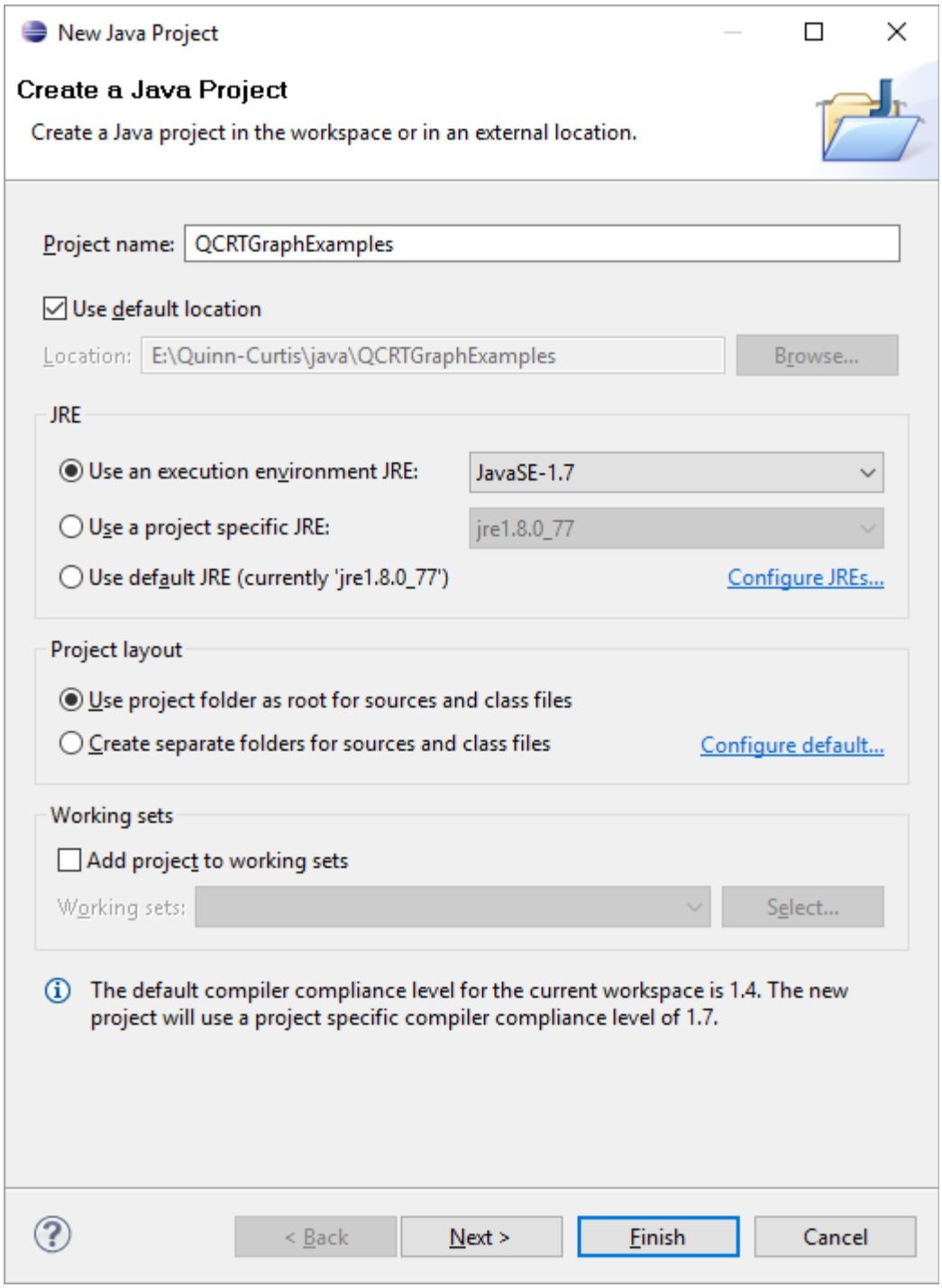
Eclipse

These directions were created using Eclipse Juno. Our standard development environment is Eclipse and our example programs are organized along the lines of the Java example programs that ship with Eclipse.

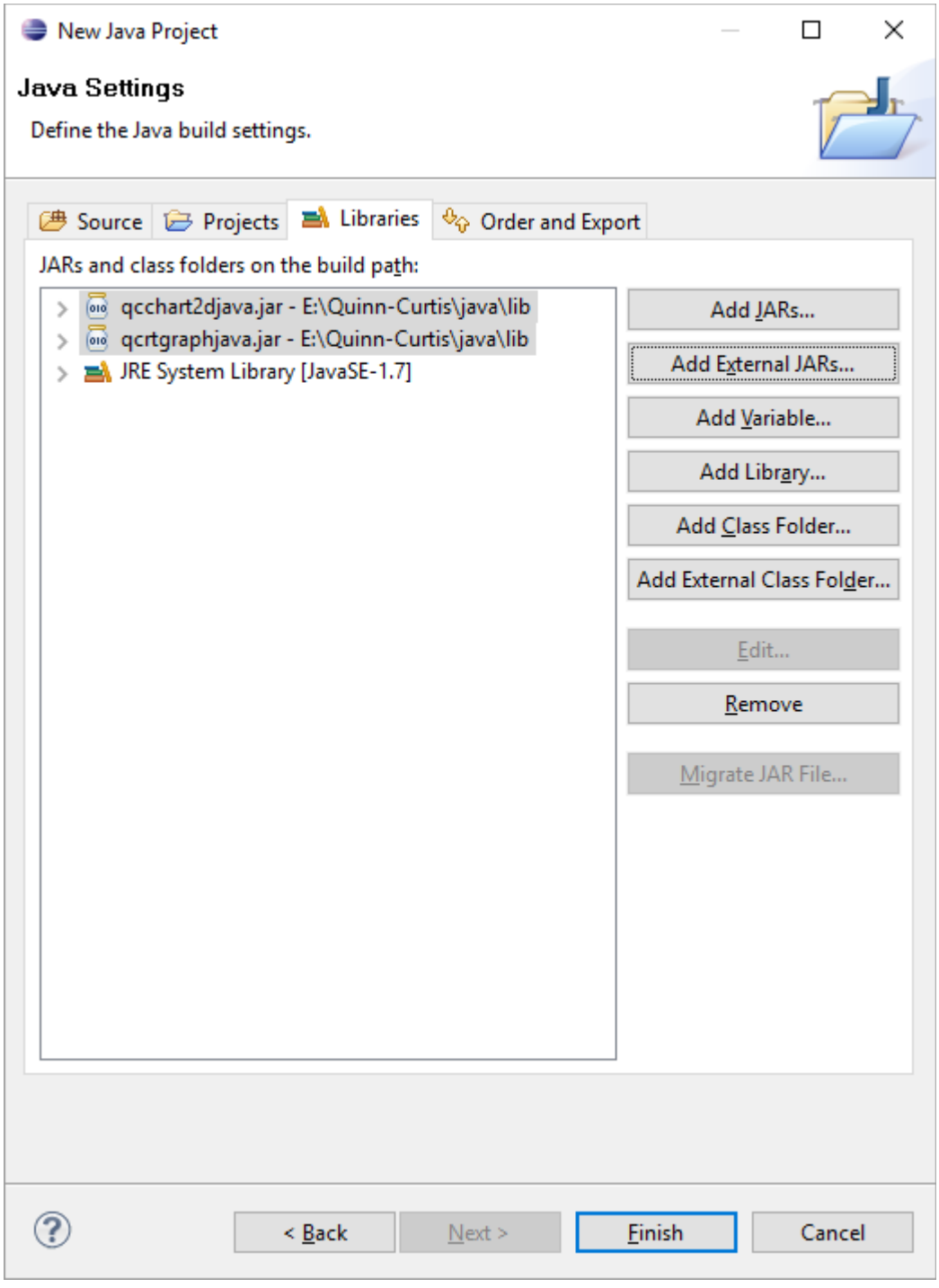
From the Eclipse Java Perspective IDE, select **File | Switch Workspace** and browse to and select the **C:\Quinn-Curtis\java** directory.



This will establish the current workspace directory as Quinn-Curtis/java. From the Eclipse main menu select **File | New | Project | Java Project.** Give the project the name QCRTGraphExamples (any name will work).). **Special Note:** We find that it works best if you check ***Use project folder as root for sources and class files*** under Project layout. This may or not be the default setting, depending on what version of Eclipse you are using.



Select **Next** to define the Java build settings. Select the **Libraries** tab and select **Add External JARs** and browse to the Quinn-Curtis/java/lib directory and add the qcchart2djava.jar and qcrtgraphjava.jar files.



All of the example programs reference the QCCChart2D and QCRTGraph for Java jar files (qcchart2djava.jar and qcrtgraphjava.jar) located in the Quinn-Curtis/java/lib folder. The qcchart2djava.jar file contains the **com.quinncurtis.chart2djava** package referenced in the

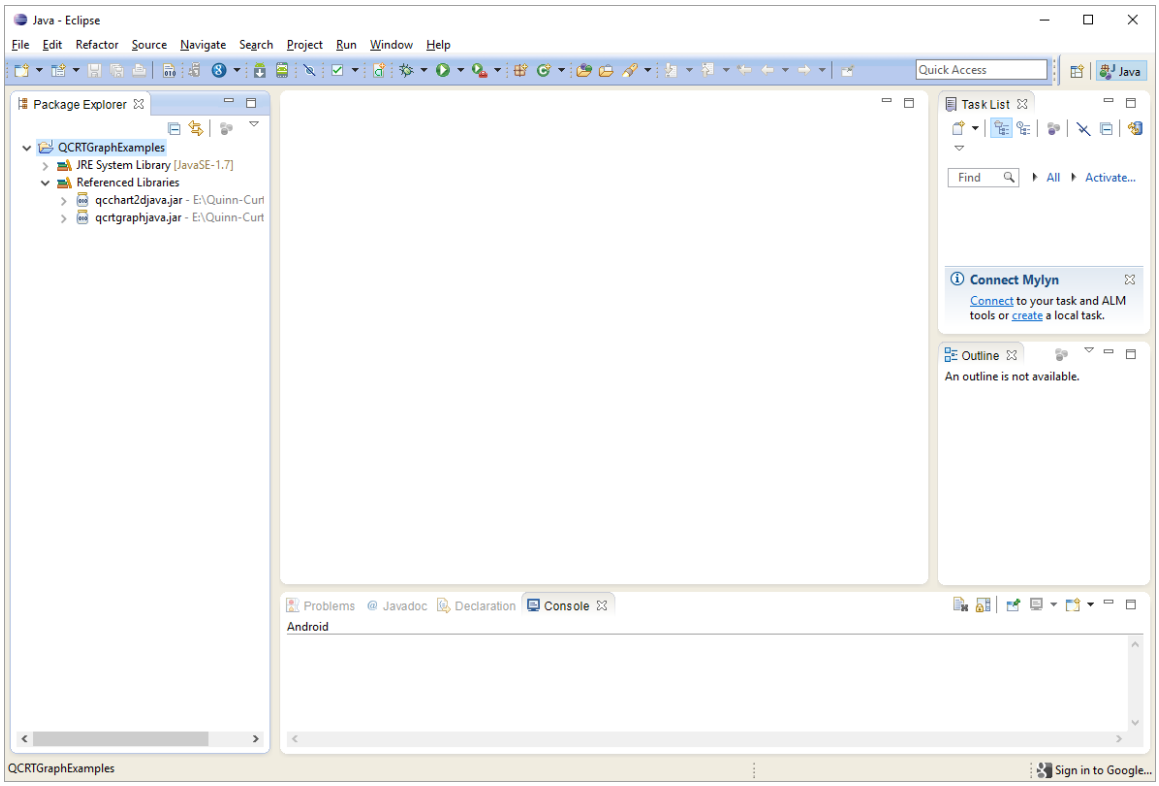
```
import com.quinncurtis.chart2djava.*;
```

statements found in all of the example program source files that contain calls to the charting routines.. The qcrtgraphjava.jar file contains the **com.quinncurtis.rtgraphjava** package referenced in the

```
import com.quinncurtis.rtgraphjava.*;
```

statements found in all of the real-time graphics example program source files.

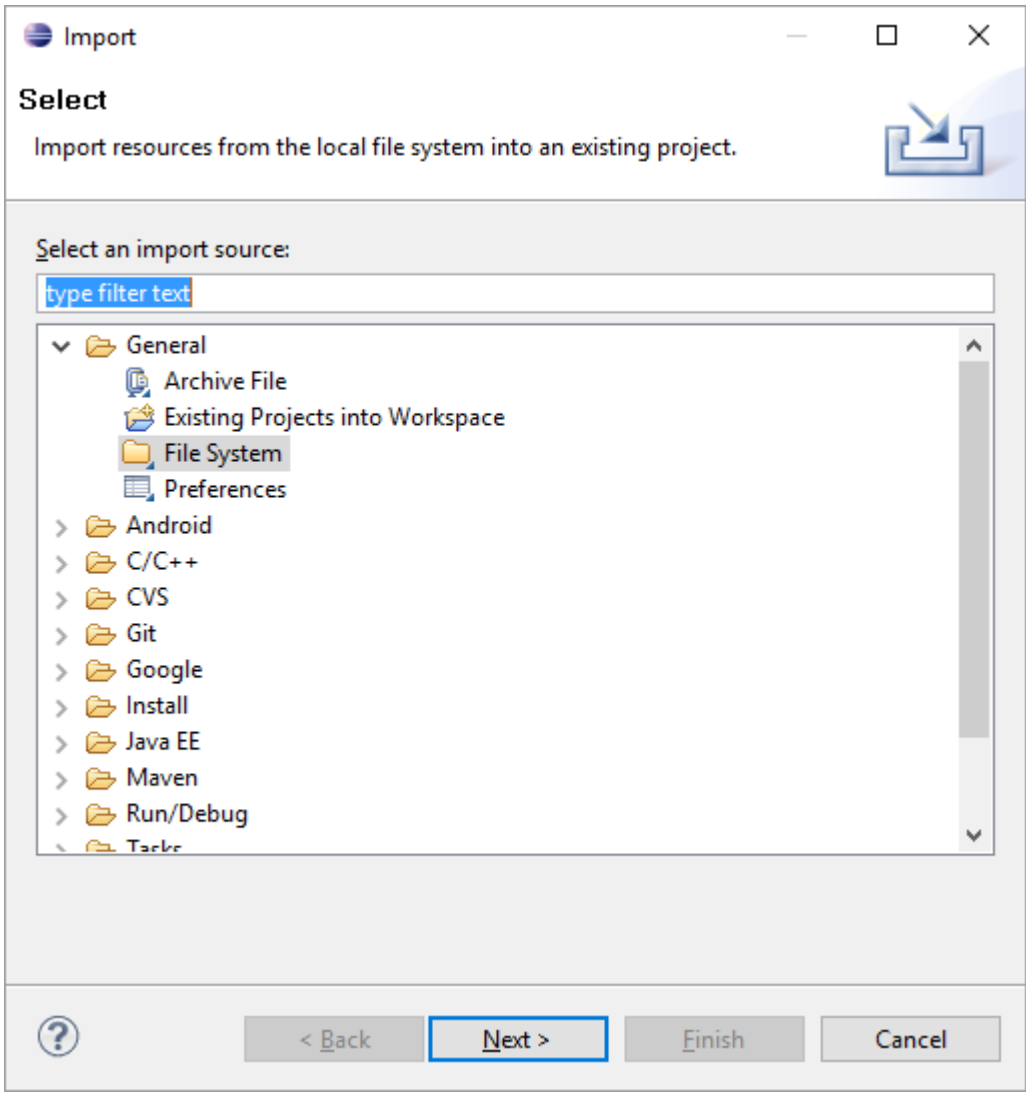
Select **Finish** to complete the creation of the basic project settings. Say Yes if it asks you for the Perspective Switch. The Package Explorer should look like:



To add the QCChart2D Javadoc documentation to the project: right click on the **qcchart2djava.jar** node and select **Properties** then **Javadoc Location**. Select the **Javadoc in Archive** radio button, and browse to qcchart2djava.jar file as the **Archive Path**, then select **doc** as the **Path within Archive**. From that point on Eclipse F1 and Shift-F2 help should be available on the qcchart2djava classes. **Repeat** the process for the **qcrtgraphjava.jar** library file.

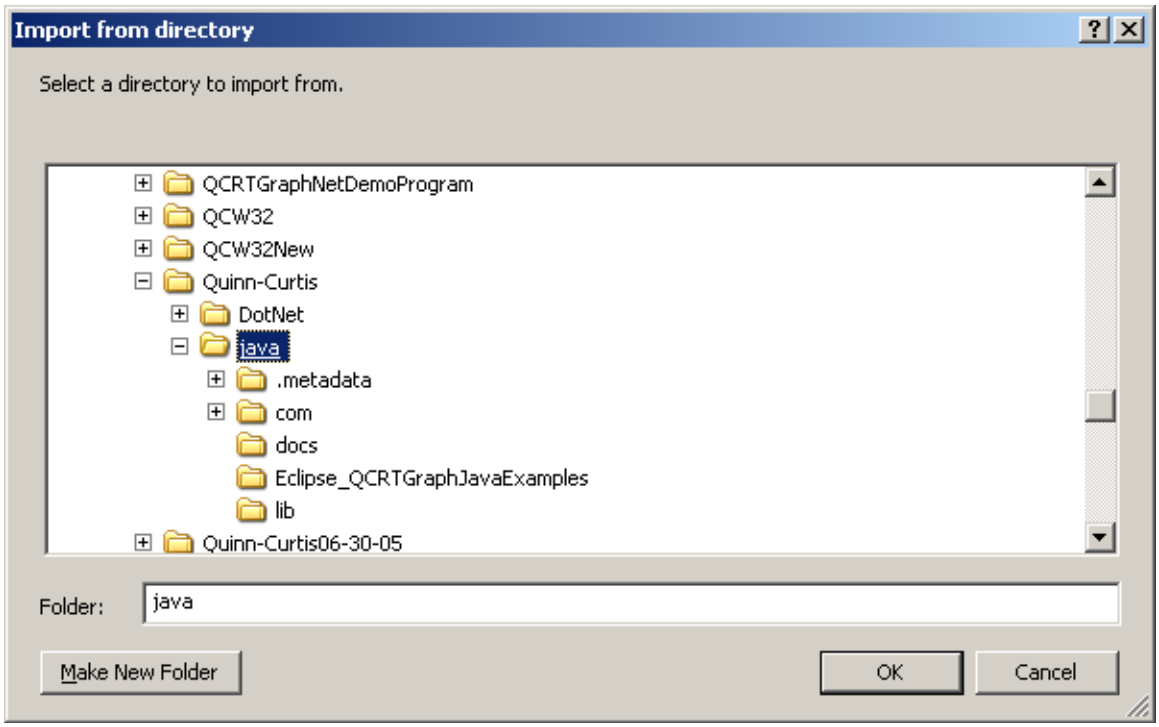
The project is still without the source files of the QCChart2D and QCRTGraph Java examples and these are added next.

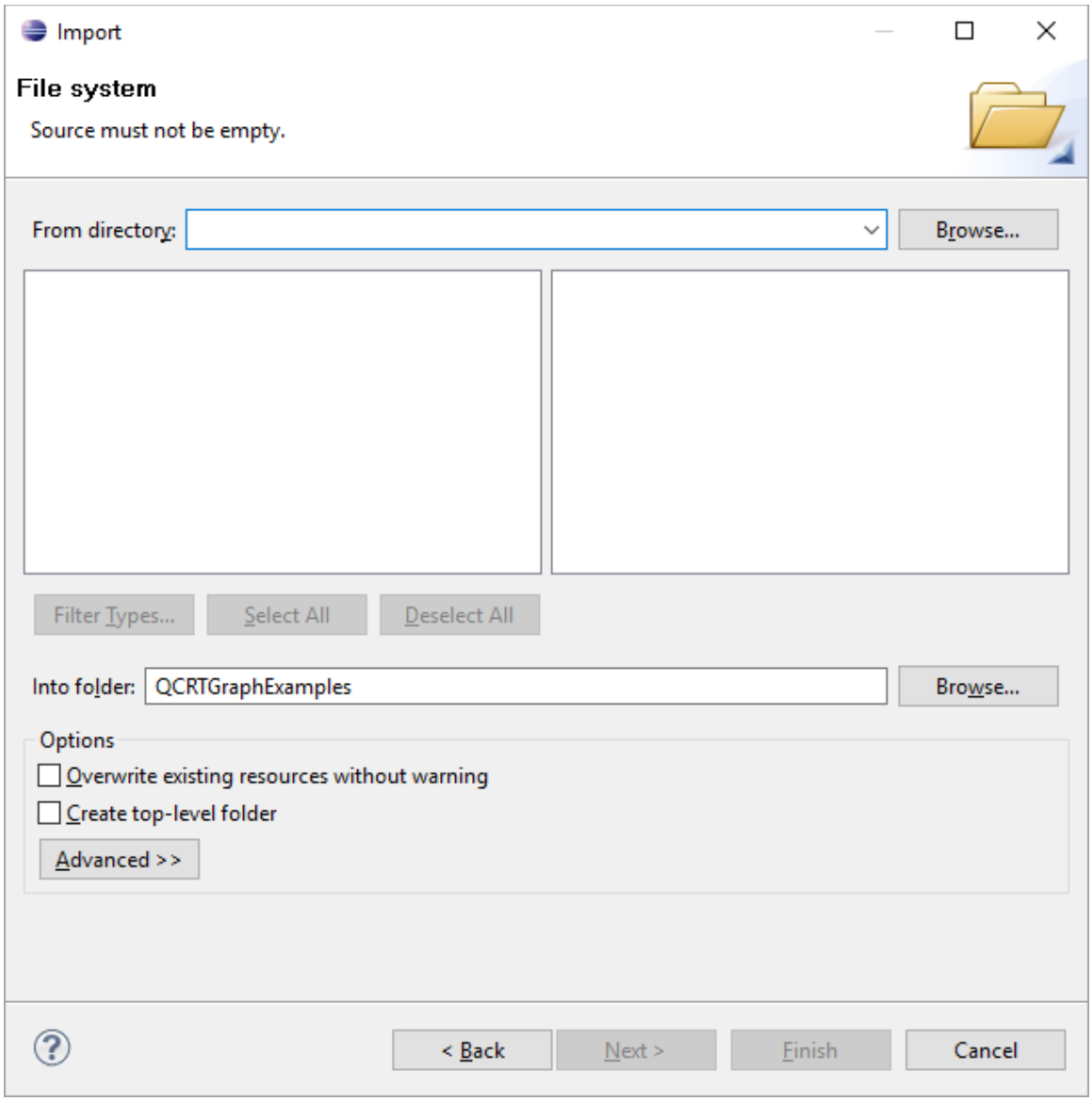
Right click the QCRTGraphExamples project in the Package Explorer and select **Import**.



The Select **File System** and Browse to the Quinn-Curtis/java directory. Select Next.

The **Into Folder** field should say QCRTGraphExamples. Select the **From Directory** Browse button and browse to the Quinn-Curtis\java folder

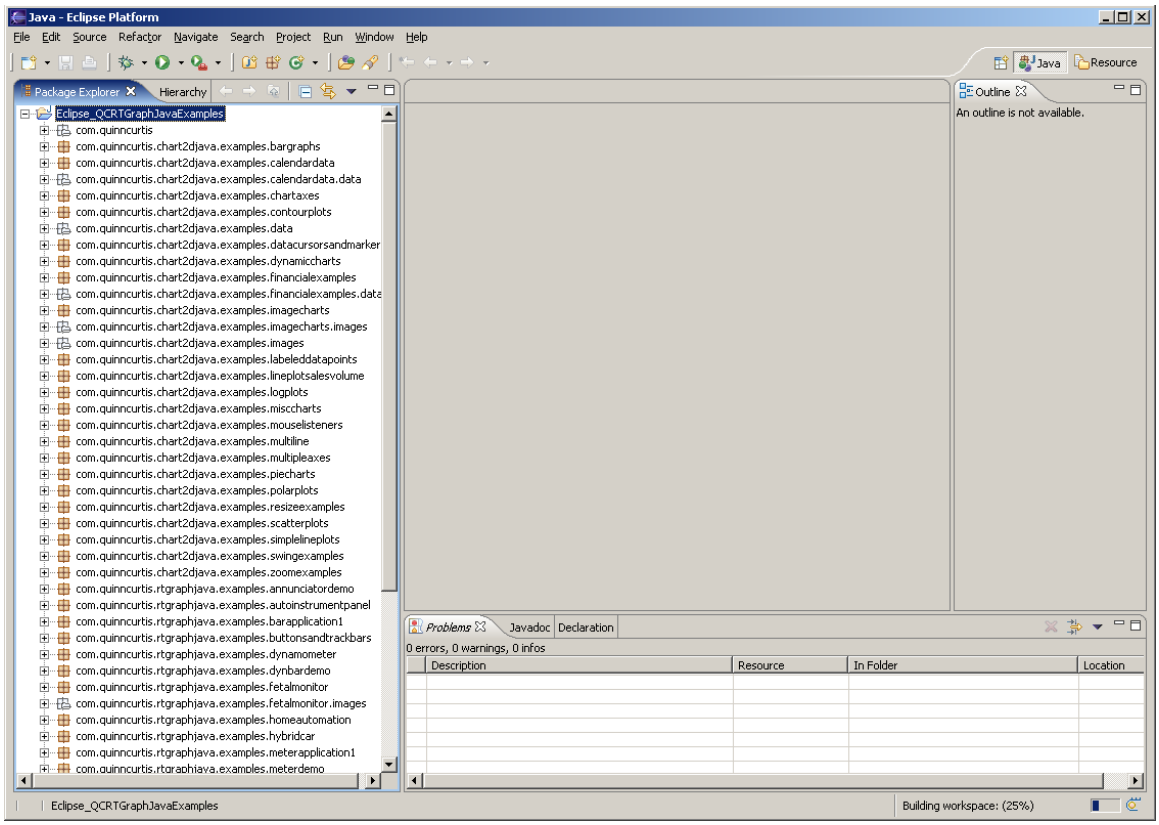

Import from directory
Select a directory to import from.
QCRTGraphNetDemoProgram
QCW32
QCW32New
Quinn-Curtis
DotNet
java
.metadata
com
docs
Eclipse_QCRTGraphJavaExamples
lib
Quinn-Curtis06-30-05
Folder:
java
Make New Folder
OK
Cancel

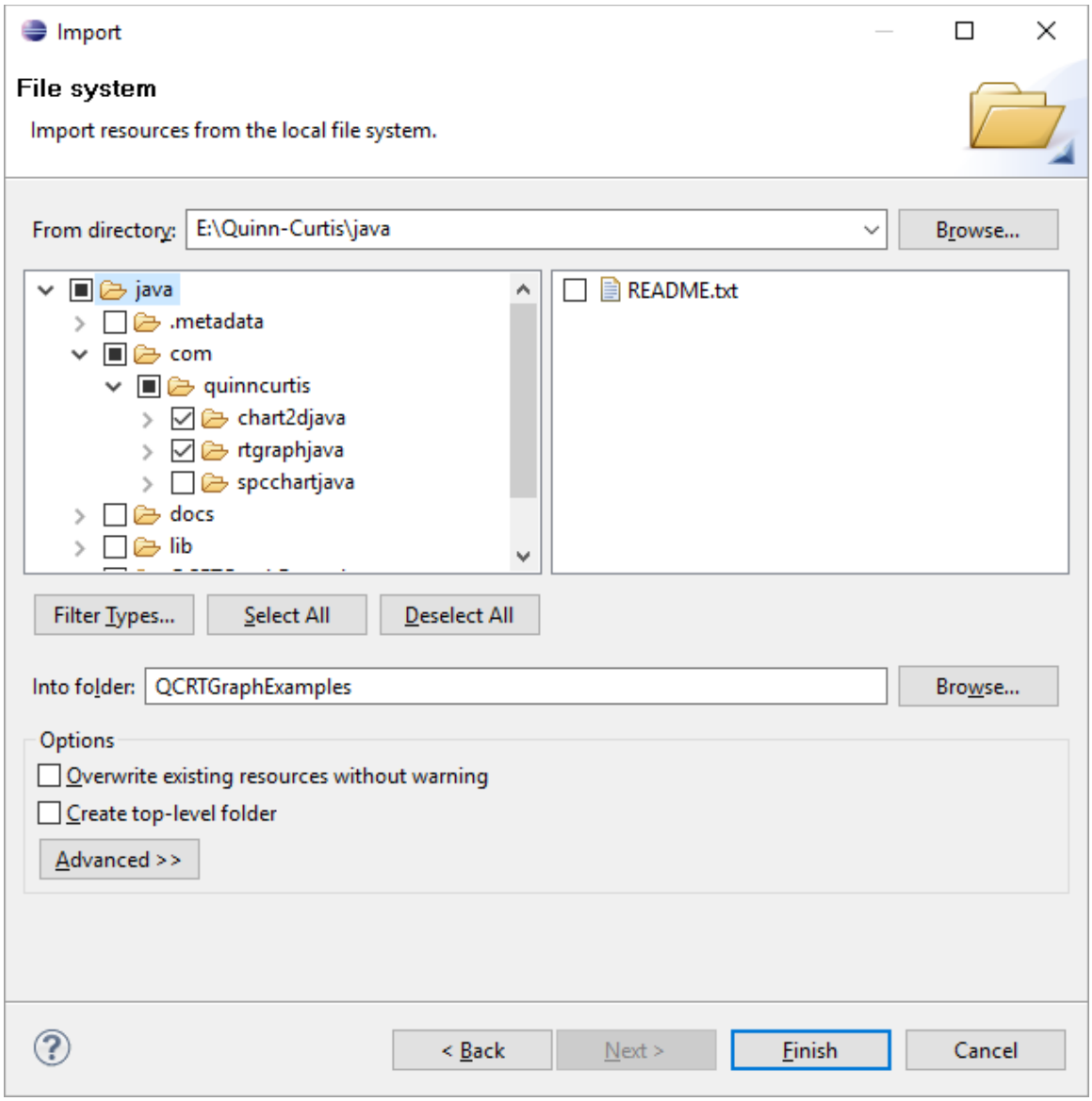


Select OK.

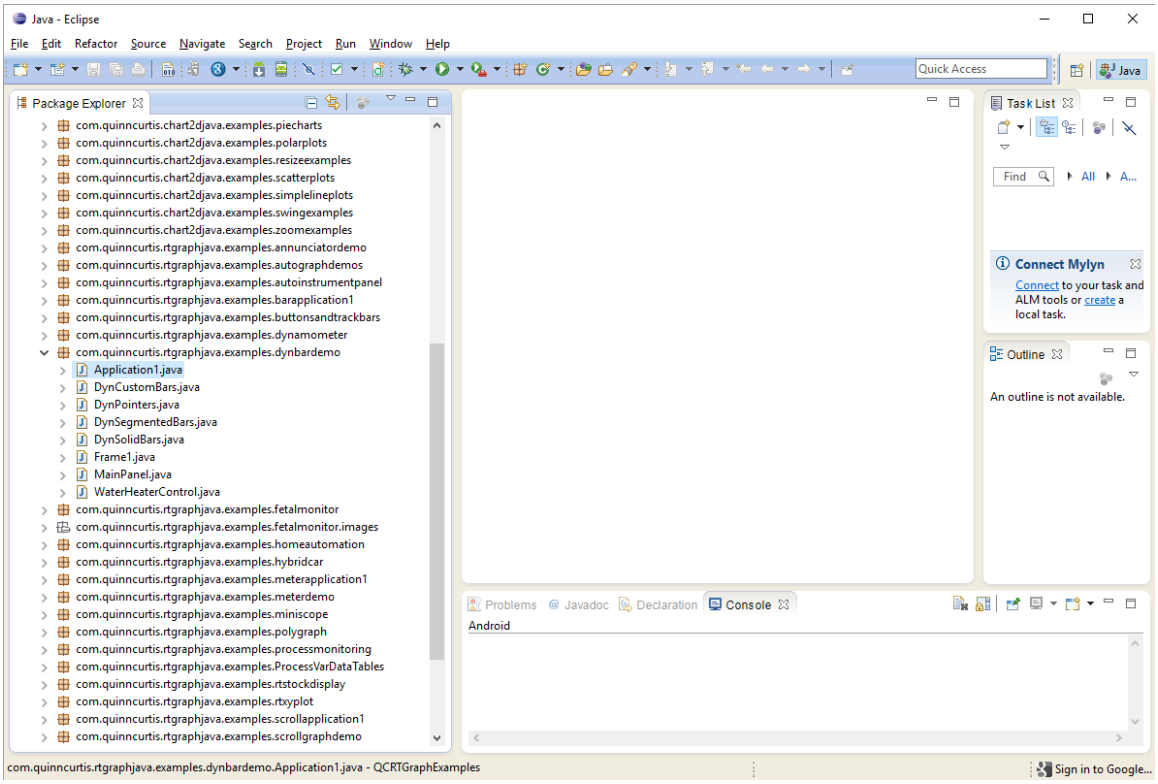
Expand the Quinn-Curtis/java directory and check the com folder.

Select **Finish** and the example program folders under com/quinncurtis will be added to the project. The entire contents of the example program folders will be copied so com/quinncurtis/ directory structure is recreated under the QCRTGraphExamples folder.


Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Resource
Package Explorer
Hierarchy
Eclipse_QCRTGraphJavaExamples
com.quinncurtis
com.quinncurtis.chart2djava.examples.bargraphs
com.quinncurtis.chart2djava.examples.calendardata
com.quinncurtis.chart2djava.examples.calendardata.data
com.quinncurtis.chart2djava.examples.chartaxes
com.quinncurtis.chart2djava.examples.contourplots
com.quinncurtis.chart2djava.examples.data
com.quinncurtis.chart2djava.examples.datacursorsandmarker
com.quinncurtis.chart2djava.examples.dynamiccharts
com.quinncurtis.chart2djava.examples.financialexamples
com.quinncurtis.chart2djava.examples.financialexamples.data
com.quinncurtis.chart2djava.examples.imagecharts
com.quinncurtis.chart2djava.examples.imagecharts.images
com.quinncurtis.chart2djava.examples.images
com.quinncurtis.chart2djava.examples.labeleddatapoints
com.quinncurtis.chart2djava.examples.lineplotsalesvolume
com.quinncurtis.chart2djava.examples.logplots
com.quinncurtis.chart2djava.examples.miscchart
com.quinncurtis.chart2djava.examples.mouselisteners
com.quinncurtis.chart2djava.examples.multiline
com.quinncurtis.chart2djava.examples.multipleaxes
com.quinncurtis.chart2djava.examples.piecharts
com.quinncurtis.chart2djava.examples.polarplots
com.quinncurtis.chart2djava.examples.resizeexamples
com.quinncurtis.chart2djava.examples.scatterplots
com.quinncurtis.chart2djava.examples.simplelineplots
com.quinncurtis.chart2djava.examples.swingexamples
com.quinncurtis.chart2djava.examples.zoomexamples
com.quinncurtis.rtgraphjava.examples.annunciatordemo
com.quinncurtis.rtgraphjava.examples.autoinstrumentpanel
com.quinncurtis.rtgraphjava.examples.barapplication1
com.quinncurtis.rtgraphjava.examples.buttonsandtrackbars
com.quinncurtis.rtgraphjava.examples.dynamometer
com.quinncurtis.rtgraphjava.examples.dynbardemo
com.quinncurtis.rtgraphjava.examples.fetalmonitor
com.quinncurtis.rtgraphjava.examples.fetalmonitor.images
com.quinncurtis.rtgraphjava.examples.homeautomation
com.quinncurtis.rtgraphjava.examples.hybridcar
com.quinncurtis.rtgraphjava.examples.meterapplication1
com.quinncurtis.rtgraphjava.examples.meterdemo
Outline
An outline is not available.
Problems
Javadoc
Declaration
0 errors, 0 warnings, 0 infos
Description
Resource
In Folder
Location
Eclipse_QCRTGraphJavaExamples
Building workspace: (25%)



Expand one of the example program folders and locate the Application1.java file. Right click on that file and select **Run | Run Java Application**.


Java - Eclipse
File Edit Refactor Source Navigate Search Project Run Window Help
Quick Access
Java
Package Explorer
com.quinncurtis.chart2djava.examples.piecharts
com.quinncurtis.chart2djava.examples.polarplots
com.quinncurtis.chart2djava.examples.resizeexamples
com.quinncurtis.chart2djava.examples.scatterplots
com.quinncurtis.chart2djava.examples.simplelineplots
com.quinncurtis.chart2djava.examples.swingexamples
com.quinncurtis.chart2djava.examples.zoomexamples
com.quinncurtis.rtgraphjava.examples.annunciatordemo
com.quinncurtis.rtgraphjava.examples.autographdemos
com.quinncurtis.rtgraphjava.examples.autoinstrumentpanel
com.quinncurtis.rtgraphjava.examples.barapplication1
com.quinncurtis.rtgraphjava.examples.buttonsandtrackbars
com.quinncurtis.rtgraphjava.examples.dynamometer
com.quinncurtis.rtgraphjava.examples.dynbardemo
Application1.java
DynCustomBars.java
DynPointers.java
DynSegmentedBars.java
DynSolidBars.java
Frame1.java
MainPanel.java
WaterHeaterControl.java
com.quinncurtis.rtgraphjava.examples.fetalmonitor
com.quinncurtis.rtgraphjava.examples.fetalmonitor.images
com.quinncurtis.rtgraphjava.examples.homeautomation
com.quinncurtis.rtgraphjava.examples.hybridcar
com.quinncurtis.rtgraphjava.examples.meterapplication1
com.quinncurtis.rtgraphjava.examples.meterdemo
com.quinncurtis.rtgraphjava.examples.miniscope
com.quinncurtis.rtgraphjava.examples.polygraph
com.quinncurtis.rtgraphjava.examples.processmonitoring
com.quinncurtis.rtgraphjava.examples.ProcessVarDataTables
com.quinncurtis.rtgraphjava.examples.rtstockdisplay
com.quinncurtis.rtgraphjava.examples.rtxyplot
com.quinncurtis.rtgraphjava.examples.scrollapplication1
com.quinncurtis.rtgraphjava.examples.scrollgraphdemo
Task List
Find
All
A...
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Outline
An outline is not available.
Problems
Javadoc
Declaration
Console
Android
com.quinncurtis.rtgraphjava.examples.dynbardemo.Application1.java - QCRTGraphExamples
Sign in to Google...

21. Tutorial – Creating QCChart2D for Java Applications and Applets

Java Applications

ScrollApplication1 for Java graphs are displayed in our **ChartView** subclass of the standard JPanel window class. Any program that you create that uses a JPanel class should be able to use our **ChartView**. You should already be a moderately experienced Java programmer before trying to use this software and should already have a collection of simple Java programs that you can experiment with. The example described below is our **ScrollApplication1** example program. The main class of the program is the **Application1** class, listed below. The **Application1** class contains the static main method that creates an instance of the **Application1** class. It creates an instance of the Frame1 class that is sized for a width and height that is half the screen width and height.

```
package com.quinncurtis.rtgraphjava.examples.scrollapplication1;

import javax.swing.UIManager;
import java.awt.*;

/**
 * Title:  scrollapplication1 example program
 * Description:  Example program for QCRTGraph for Java  real-time graphics
software
 * Copyright:  Copyright (c) 2005
 * Company:  Quinn-Curtis, Inc.
 * @author:  Staff
 * @version  1.5
 */

public class Application1 {
  boolean packFrame = false;

  //Construct the application
  public Application1() {
    Frame1 frame = new Frame1();
    //Validate frames that have preset sizes
    //Pack frames that have useful preferred size info, e.g. from their layout
    if (packFrame) {
      frame.pack();
    }
    else {
      frame.validate();
    }
    //Center the window
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = frame.getSize();
    if (frameSize.height > screenSize.height) {
      frameSize.height = screenSize.height;
    }
    if (frameSize.width > screenSize.width) {
      frameSize.width = screenSize.width;
    }
    frame.setLocation((screenSize.width - frameSize.width) / 2, (screenSize.height
- frameSize.height) / 2);
    frame.setVisible(true);
  }
```

```
  //Main method
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    new Application1();
  }
}
```

The **Frame1** class is derived from the Java **JFrame** class and provides a framework for combining menus and content panels. In this example it combines a simple menu bar with the **ScrollApplication1** class, a **ChartView** derived graph. The borderLayout1 layout manager positions the menu bar at the top of the frame, and the **ScrollApplication1** class in the center of the frame.

```
package com.quinncurtis.rtgraphjava.examples.scrollapplication1;
import java.awt.*;
import java.awt.event.*;
import java.io.InputStream;
import javax.swing.*;
import com.quinncurtis.chart2djava.*;


/**
 * Title:  scrollapplication1 example program
 * Description:  Example program for QCRTGraph for Java  real-time graphics
software
 * Copyright:  Copyright (c) 2005
 * Company:  Quinn-Curtis, Inc.
 * @author:  Staff
 * @version  1.5
 */

public class Frame1 extends JFrame {
  JPanel contentPane;
  BorderLayout borderLayout1 = new BorderLayout();

  // Menu stuff
  JMenuBar jmenubar = new JMenuBar();
  JMenu jMenu1 = new JMenu();
  JMenuItem printMenuItem = new JMenuItem();
  JMenuItem printerSetupMenuItem = new JMenuItem();
  JMenuItem imageSetupMenuItem = new JMenuItem();

  JMenu jMenu2 = new JMenu();
  JMenuItem exitMenuItem = new JMenuItem();

  // MenuListener class found at bottom of this file
  MenuListener menulistener = new MenuListener();

  ScrollApplication1 scrollapplication1;
  ChartPrint printgraph = new ChartPrint();


  //Construct the frame
  public Frame1() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      initFrame();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
```

```
  private void setupMenu() {
    // Setup menu bar
    jMenu1.setText("File");
    printMenuItem.setText("Print Graph");
    printMenuItem.addActionListener(menulistener);

    printerSetupMenuItem.setText("Printer Setup");
    printerSetupMenuItem.addActionListener(menulistener);

    imageSetupMenuItem.setText("Save As JPEG");
    imageSetupMenuItem.addActionListener(menulistener);

    jMenu2.setText("Exit");
    exitMenuItem.setText("Exit");
    exitMenuItem.addActionListener(menulistener);

    jmenubar.add(jMenu1);
    jmenubar.add(jMenu2);
    jMenu1.add(printMenuItem);
    jMenu1.add(printerSetupMenuItem);
    jMenu1.add(imageSetupMenuItem);
    jMenu2.add(exitMenuItem);
    // add menu to frame
    contentPane.add(jmenubar, BorderLayout.NORTH);
  }



  //Component initialization
  private void initFrame() throws Exception  {
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout1);

    this.setSize(new Dimension(1000, 800));

    // Setup menu bar
    setupMenu();

    scrollapplication1 = new ScrollApplication1();

    contentPane.add(scrollapplication1, BorderLayout.CENTER);

  }


  //Overridden so we can exit when window is closed
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }

  class MenuListener implements ActionListener {
,
,
,

}

}
```

The **ScrollApplication1** class is where the chart is actually defined. Since it makes calls into the **com.quinncurtis.chart2djava** and **com.quinncurtis.rtgraphjava** packages, those package needs to be referenced in the import section of the class. The majority of the manual describes how to create and interact with the chart objects in the **com.quinncurtis.chart2djava** and **com.quinncurtis.rtgraphjava** packages.

```
package com.quinncurtis.rtgraphjava.examples.scrollapplication1;

import java.awt.*;
import java.awt.event.*;
import java.text.SimpleDateFormat;
import java.util.*;
import com.quinncurtis.chart2djava.*;
import com.quinncurtis.rtgraphjava.*;

/**
 * @author Quinn-Curtis Staff
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class ScrollApplication1 extends ChartView implements RTAlarmEventListener
{

    double  currentTemperatureValue1 = 110.0;
    RTProcessVar currentTemperature1;
    double  currentTemperatureValue2 = 100.0;
    RTProcessVar currentTemperature2;
    RTScrollFrame scrollFrame = new RTScrollFrame();

    RTAlarm templowalarm1;
    RTAlarm temphighalarm1;
    RTAlarm templowalarm2;
    RTAlarm temphighalarm2;

    int counter = 0;
    Font font12 = new Font("SansSerif",Font.PLAIN, 12);
    Font font14 = new Font("SansSerif",Font.PLAIN, 14);
    Font font14Numeric = new Font("Digital SF", Font.PLAIN,14);


    Color bisque = new Color(0xFFE4C4);

      javax.swing.Timer eventTimer1 = new javax.swing.Timer(300, new
ActionListener()  {
          public void actionPerformed(ActionEvent e) {
            timer1_Tick(e);
          }
        });

      javax.swing.Timer eventTimer2 = new javax.swing.Timer(1000, new
ActionListener()  {
          public void actionPerformed(ActionEvent e) {
            timer2_Tick(e);
          }
        });

      public ScrollApplication1() {
        try {
            InitializeGraph();
        }
        catch(Exception e) {
          e.printStackTrace();
        }
      }

        private void InitializeScrollGraph()
        {
            double x1 = 0.1;
            double y1 = 0.1;
            double x2 = 0.90;
            double y2 = 0.9;
            Font axisFont = font12;
```

```
            ChartView chartVu  = this;
            GregorianCalendar startTime = new GregorianCalendar();
            GregorianCalendar endTime = new GregorianCalendar();

            endTime.add(ChartCalendar.SECOND,60);

            TimeCoordinates pTransform1 = new TimeCoordinates( startTime,
currentTemperature1.getDefaultMinimumDisplayValue(),
                endTime, currentTemperature1.getDefaultMaximumDisplayValue());

            pTransform1.setGraphBorderDiagonal(x1, y1, x2, y2) ;

            Background graphbackground = new Background( pTransform1,
ChartConstants.GRAPH_BACKGROUND, Color.white);
            chartVu.addChartObject(graphbackground);

            Background plotbackground = new Background( pTransform1,
ChartConstants.PLOT_BACKGROUND, Color.black);
            chartVu.addChartObject(plotbackground);

            TimeAxis xaxis = new TimeAxis(pTransform1, ChartConstants.X_AXIS);
            xaxis.setLineColor( Color.black);
            chartVu.addChartObject(xaxis);

            LinearAxis yaxis = new LinearAxis(pTransform1, ChartConstants.Y_AXIS);
            yaxis.setLineColor( Color.black);
            chartVu.addChartObject(yaxis);

            Grid yAxisGrid = new Grid(xaxis, yaxis, ChartConstants.Y_AXIS,
ChartConstants.GRID_MAJOR);
            yAxisGrid.setColor(Color.white);
            yAxisGrid.setLineWidth(1);
            yAxisGrid.setLineStyle(ChartConstants.LS_DOT_1_4);
            chartVu.addChartObject(yAxisGrid);

            Grid xAxisGrid = new Grid(xaxis, yaxis, ChartConstants.X_AXIS,
ChartConstants.GRID_MAJOR);
            xAxisGrid.setColor(Color.white);
            xAxisGrid.setLineWidth(1);
            xAxisGrid.setLineStyle(ChartConstants.LS_DOT_1_4);
            chartVu.addChartObject(xAxisGrid);

            LinearAxis yaxis2 = new LinearAxis(pTransform1, ChartConstants.Y_AXIS);
            yaxis2.setLineColor( Color.black);
            yaxis2.setAxisTickDir( ChartConstants.AXIS_MAX);
            yaxis2.setAxisIntercept(xaxis.getAxisMax());
            chartVu.addChartObject(yaxis2);

            TimeAxisLabels xAxisLab = new TimeAxisLabels(xaxis);
            xAxisLab.setTextFont( axisFont);
            chartVu.addChartObject(xAxisLab);

            NumericAxisLabels yAxisLab = new NumericAxisLabels(yaxis);
            yAxisLab.setTextFont( axisFont);
            chartVu.addChartObject(yAxisLab);

            scrollFrame = new RTScrollFrame(this, currentTemperature1, pTransform1,
ChartConstants.RT_FIXEDEXTENT_MOVINGSTART_AUTOSCROLL);
            scrollFrame.addProcessVar(currentTemperature2);

            scrollFrame.setScrollScaleModeY( ChartConstants.RT_AUTOSCALE_Y_MINMAX);
            // Allow 100 samples to accumlate before autoscaling y-axis. This
prevents rapid
            // changes of the y-scale for the first few samples
            scrollFrame.setMinSamplesForAutoScale ( 100);
            scrollFrame.setScrollRescaleMargin( 0.05);
            chartVu.addChartObject(scrollFrame);

            ChartAttribute attrib1 = new ChartAttribute (Color.yellow, 3,
ChartConstants.LS_SOLID);
```

```
            SimpleLinePlot lineplot1 =  new SimpleLinePlot(pTransform1, null,
attrib1);
            lineplot1.setFastClipMode( ChartConstants.FASTCLIP_X);
            RTSimpleSingleValuePlot solarPanelLinePlot1 = new
RTSimpleSingleValuePlot(pTransform1,lineplot1, currentTemperature1);
            chartVu.addChartObject(solarPanelLinePlot1);

            ChartAttribute attrib2 = new ChartAttribute (Color.green,
3,ChartConstants.LS_SOLID);
            SimpleLinePlot lineplot2 =  new SimpleLinePlot(pTransform1, null,
attrib2);
            lineplot2.setFastClipMode( ChartConstants.FASTCLIP_X);
            RTSimpleSingleValuePlot solarPanelLinePlot2 = new
RTSimpleSingleValuePlot(pTransform1,lineplot2, currentTemperature2);
            chartVu.addChartObject(solarPanelLinePlot2);

            RTAlarmIndicator alarmlines = new RTAlarmIndicator(yaxis,
solarPanelLinePlot1);


alarmlines.setAlarmIndicatorMode( ChartConstants.RT_ALARM_LIMIT_LINE_INDICATOR);
            // Auto-scale can move alarm lines outside of plot area, clip the lines
in this case.
            alarmlines.setChartObjClipping ( ChartConstants.PLOT_AREA_CLIPPING);
            chartVu.addChartObject(alarmlines);

            Font titlefont = font14;
            ChartTitle charttitle = new ChartTitle(pTransform1, titlefont,"Scroll
Application #1", ChartConstants.CHART_HEADER, ChartConstants.CENTER_PLOT);
            chartVu.addChartObject(charttitle);

        }



        public void  InitializeGraph()
        {

            templowalarm1 = new RTAlarm(ChartConstants.RT_ALARM_LOWERTHAN, 80);
            templowalarm1.setAlarmMessage( "Low Alarm");
            templowalarm1.setAlarmSymbolColor(Color.blue);
            templowalarm1.setAlarmTextColor(Color.blue);

            temphighalarm1 = new RTAlarm(ChartConstants.RT_ALARM_GREATERTHAN, 120);
            temphighalarm1.setAlarmMessage("High Alarm");
            temphighalarm1.setAlarmSymbolColor(Color.red);
            temphighalarm1.setAlarmTextColor(Color.red);

            currentTemperature1 = new RTProcessVar("Temp 1", new
ChartAttribute(Color.green, 1.0, ChartConstants.LS_SOLID, Color.green));
            currentTemperature1.setMinimumValue(-20);
            currentTemperature1.setMaximumValue(200);
            currentTemperature1.setDefaultMinimumDisplayValue(0);
            currentTemperature1.setDefaultMaximumDisplayValue(150);
            currentTemperature1.setCurrentValue( currentTemperatureValue1);
            currentTemperature1.addAlarm(templowalarm1);
            currentTemperature1.addAlarm(temphighalarm1);
            // Important, enables historical data collection for scroll graphs
            currentTemperature1.setDatasetEnableUpdate(true);
            currentTemperature1.addAlarmTransitionEventListener(this);
            currentTemperature1.setAlarmTransitionEventEnable(true);


            templowalarm2 = new RTAlarm(ChartConstants.RT_ALARM_LOWERTHAN, 80);
            templowalarm2.setAlarmMessage( "Low Alarm");
            templowalarm2.setAlarmSymbolColor(Color.blue);
            templowalarm2.setAlarmTextColor(Color.blue);

            temphighalarm2 = new RTAlarm(ChartConstants.RT_ALARM_GREATERTHAN, 120);
            temphighalarm2.setAlarmMessage("High Alarm");
            temphighalarm2.setAlarmSymbolColor(Color.red);
```

```
            temphighalarm2.setAlarmTextColor(Color.red);

            currentTemperature2 = new RTProcessVar("Temp 2", new
ChartAttribute(Color.green, 1.0, ChartConstants.LS_SOLID, Color.green));
            currentTemperature2.setMinimumValue(-20);
            currentTemperature2.setMaximumValue(200);
            currentTemperature2.setDefaultMinimumDisplayValue(0);
            currentTemperature2.setDefaultMaximumDisplayValue(150);
            currentTemperature2.setCurrentValue( currentTemperatureValue2);
            currentTemperature2.addAlarm(templowalarm2);
            currentTemperature2.addAlarm(temphighalarm2);
            // Important, enables historical data collection for scroll graphs
            currentTemperature2.setDatasetEnableUpdate(true);
            currentTemperature2.addAlarmTransitionEventListener(this);
            currentTemperature2.setAlarmTransitionEventEnable(true);

            InitializeScrollGraph();

            eventTimer1.start();
            eventTimer2.start();

        }

        // This method is required to resolve the requirement of the
RTAlarmEventListener interface
        public void AlarmEventChanged(RTProcessVar sender, RTAlarmEventArgs e)
        {  RTAlarm alarm = e.getEventAlarm();
           double alarmlimitvalue = alarm.getAlarmLimitValue();
           String alarmlimitstring = Double.toString(alarmlimitvalue);
           RTProcessVar pv = e.getProcessVar();
           String tagname = pv.getTagName();
           GregorianCalendar timestamp = pv.getCalendarTimeStamp();
           SimpleDateFormat formatter = new SimpleDateFormat ("hh:mm:ss.SS");
           String timestampstring = formatter.format(timestamp.getTime());
            if (alarm.getAlarmState())
              System.out.println(timestampstring + " " + tagname + " Warning -
Alarm Level " + alarmlimitstring + " Exceeded");
            else
              System.out.println(timestampstring + " " + tagname + " Process Value
transitioned back to normal range.");

        }


        // Update data using 200 msec timer
        private void timer1_Tick(ActionEvent e)
        {

            // Random data
            currentTemperatureValue1 += 5 * (0.5 - ChartSupport.getRandomDouble());
            currentTemperatureValue2 += 8 * (0.5 - ChartSupport.getRandomDouble());

            // These two methods of setting the current value are equivalent

            // This method uses the default time stamp, which is the current time-
of-day
            currentTemperature1.setCurrentValue(currentTemperatureValue1);
            currentTemperature2.setCurrentValue(currentTemperatureValue2);

            // This method you pass in whatever time stamp you want, in this case
it is the current time-of-day
//          GregorianCalendar tod = new GregorianCalendar();  // get current time
//          currentTemperature1.setCurrentValue(tod, currentTemperatureValue1);
//          currentTemperature1.setCurrentValue(tod, currentTemperatureValue2);

        }

        // Update screen on 1000 msec timer
        private void timer2_Tick(ActionEvent e)
        {
            this.updateDraw();
```

```
        }

          public void Print(ChartPrint printobj)
          {
            printobj.setPrintChartView(this);
            printobj.startPrint();
          }

         public void SaveJPEG()
          { String filename = "ScrollApplication1.jpg";
            ChartBufferedImage savegraphJPEG = new ChartBufferedImage();
            ImageFileChooser imagefilechooser = new ImageFileChooser(this);
            if (imagefilechooser.process(filename))
            {
            filename = imagefilechooser.getSelectedFilename();
            savegraphJPEG.setChartViewComponent(this);
            savegraphJPEG.render();
            savegraphJPEG.saveImageAsJPEG(filename);
            }
          }

}
```

That is the complete program. All of the other programs are setup in a similar way. Most of the other example programs place a **JTabbedPanel** in the main content panel of the frame and then place individual charts as tabs in the **JTabbedPanel**.

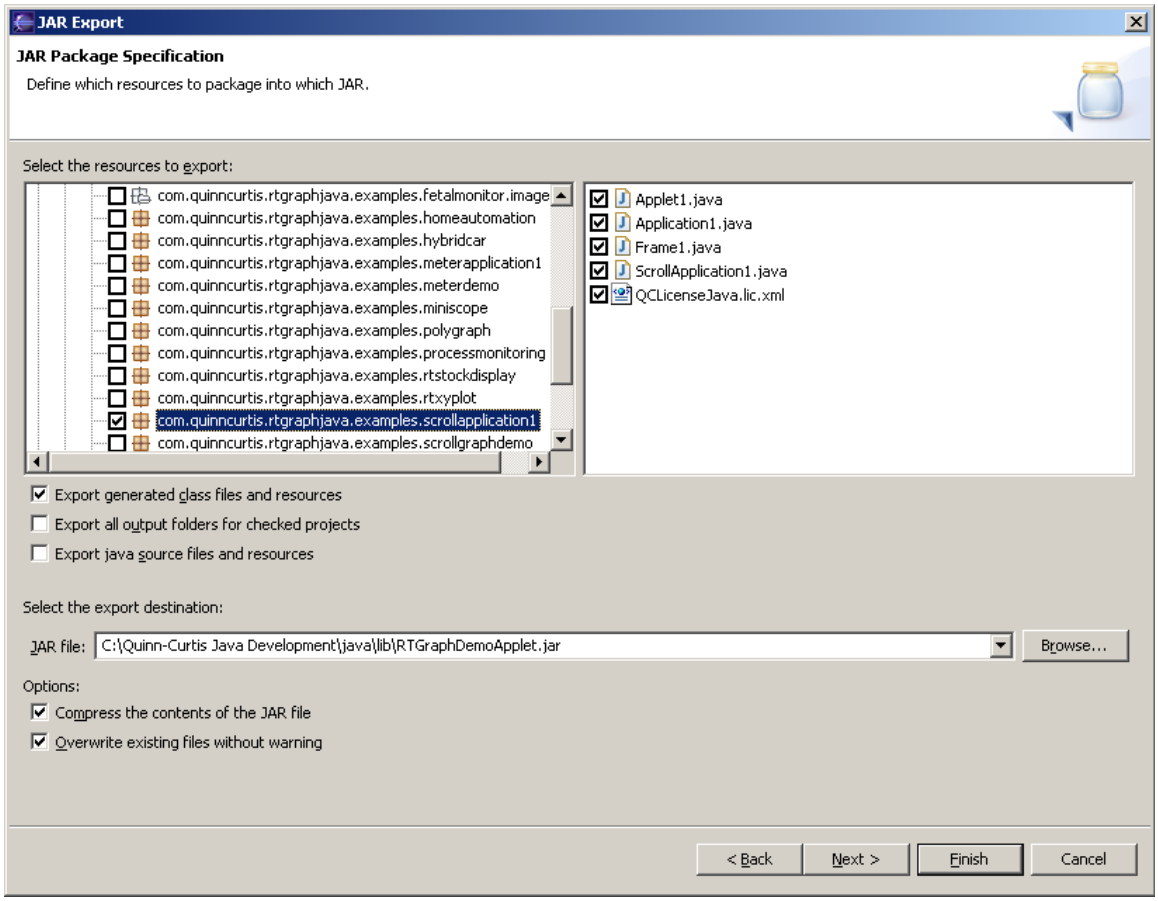
Java Applets

While Java Applications are useful if you are creating a program to run as a standalone application on a desktop, Java Applets are used to create Java programs that run in a web browser. The applet below creates the same **ScrollApplication1** chart as in the previous application. The source for the Applet version is also found in the **ScrollApplication1** example program directory. The **JFrame** window used in the Application example is not used here, since that would force the graph to be created in a window separate from the browser. Instead, the **ScrollApplication1** graph is created in the content pane of the Applet window.

```
package com.quinncurtis.rtgraphjava.examples.scrollapplication1;
import java.awt.*;
import java.io.InputStream;
import com.quinncurtis.chart2djava.ChartView;

 public class Applet1 extends javax.swing.JApplet {

    public void init() {
        initDemo();
    }

    public void start() {

    }

    public void initDemo()
    {

        // This skips the frame window and places the MainPanel directly
```

```
        // in the content pane of the Applet
        Container contentPane = this.getContentPane();
        ScrollApplication1 scrollapplication1 = new ScrollApplication1();
        contentPane.add(scrollapplication1, BorderLayout.CENTER);
    }
}
```

The program above can be run as-is in an Applet viewer. What you really want to do is to run the program from a web browser. To do that you need to create a jar file of the applet and add the proper initialization code to an HTML page viewable from your web site. Using Eclipse you would select the File | Export option. Select the files and resources in the ScrollApplication1 directory, including any other data files and resources you are using.



In this case the jar file is saved using the name scrollapplication1.jar. This is one of the three jars you will need in order to run the application in a web browser. The other jar files you will need are the qcchart2djava.jar and qcrtgraphjava.jar files found in the \quinn-curtis\java\lib directory. These jar files are uploaded to your server. The code that needs to be added to your web page will look like:

```
<APPLET CODE =
"com.quinncurtis.rtgraphjava.examples.scrollapplication1.Applet1.class"
         CODEBASE = "http://quinn-curtis.com/classes/"
         ARCHIVE = "scrollapplication1.jar, qcchart2djava.jar, qcrtgraphjava.jar"
WIDTH = 800 HEIGHT = 600>
  This browser does not support Java 1.2 (or greater) applets !
```

```
</APPLET> </p>
```

This HTML page (ScrollApplication1Applet1.htm) can be loaded and the applet run at our web site using the following link:

http://quinn-curtis.com/ScrollApplication1Applet1.htm

Note the following:

- The CODE attribute specifies the full namespace of the main applet class, com.quinn-curtis.rtgraphjava.examples.scrollapplication1.Applet1.class
- The ARCHIVE attributer specifies three jars, one that contains the code of the applet (scrollapplication1.jar), and the other two the QCChart2D (qcchart2djava.jar) and QCRTGraph (qcrtgraphjava.jar) libraries.
- The CODEBASE attribute tells the browser in which directory to find the applet. If the applet is in the same directory as the calling HTML page the CODEBASE attribute is not necessary.
- The HEIGHT and WIDTH attribute specify the pixel dimensions of the window the applet will display in

This is the simple version of enabling an applet in a web page. The more complicated way, and the way recommended by Sun is to use the HTML <OBJECT> tag. Apparently the <APPLET> tag has been deprecated by the <OBJECT> tag in the HTML standard and Sun is encouraging web designers to use it. You can read all about it from the source at:
http://docsun.cites.uiuc.edu/sun_docs/C/solaris_9/SUNWaadm/JVPLUGINUG/p5.html

They have a program that will convert <APPLET> code to <OBJECT> code, and forces the browser to download a Java plug-in if one is not already present on the computer. It also uses the EMBED tag instead of OBJECT if the Netscape browser is used. The Netscape trick involves the use of the <COMMENT> tag. Read about that in the Sun article hyperlinked above. The converted version of the HTML applet code looks like.

```
<OBJECT
    classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
    WIDTH = 800 HEIGHT = 600
    codebase="http://java.sun.com/jpi/jinstall-14-win32.cab#Version=1,4,0,mn">
    <PARAM NAME = "code" VALUE =
"com.quinncurtis.rtgraphjava.examples.scrollapplication1.Applet1.class" >
    <PARAM NAME="codebase" VALUE="http://quinn-curtis.com/classes/" >
    <PARAM NAME = "archive" VALUE = "scrollapplication1.jar, qcchart2djava.jar,
qcrtgraphjava.jar" >
    <PARAM NAME="type" VALUE="application/x-java-applet;version=1.4">
    <PARAM NAME="scriptable" VALUE="false">
```

```
    <COMMENT>
    <EMBED
            type="application/x-java-applet;version=1.4"
            CODE =
"com.quinncurtis.rtgraphjava.examples.scrollapplication1.Applet1.class"
            CODEBASE = "http://quinn-curtis.com/classes/"
            ARCHIVE = "scrollapplication1.jar, qcchart2djava.jar,
qcrtgraphjava.jar"
            WIDTH = 800
            HEIGHT = 600
        scriptable=false
        pluginspage="http://java.sun.com/jpi/plugin-install.html">
            <NOEMBED>
       No Java 2 SDK, Standard Edition v 1.4 support for APPLET!!
    </NOEMBED>
    </EMBED>
    </COMMENT></OBJECT>
>
```

This HTML page ScrollApplication1Applet2.htm) can be loaded and the applet run at our web site using the following link:

http://quinn-curtis.com/ScrollApplication1Applet2.htm

Note that while similar, some of the attributes and tags have a different use. The best place to read about the differences is in the Sun article referenced at http://docsun.cites.uiuc.edu/sun_docs/C/solaris_9/SUNWaadm/JVPLUGINUG/p5.html

The complete source to the HTML pages are found in the ScrollApplication1 examples directory, file ScrollApplication1Applet1.htm and ScrollApplication1Applet2.htm).

Note the following:

- The main applet class, com.quinn-curtis.rtgraphjava.examples.scrollapplication1.Applet1.class is specified using the "code" <PARAM>
- The required jar files are specified using the "archive" <PARAM>.
- The directory that the applet is in is specified using the "codebase" <PARAM>. If the applet is in the same directory as the calling HTML page the "codebase" <PARAM> is not necessary.
- The HEIGHT and WIDTH attribute specify the pixel dimensions of the window the applet will display in
- The rest of the tags and <PARAM> values support the automatic download of the Sun Java plug-in, if the client viewing the applet needs it.
- The <EMBED> section implements an equivalent block of HTML code that is only called if a Netscape browser is used to view the applet.

.

22. Frequently Asked Questions

FAQs

First, read the FAQ's section of the **QCChart2D** manual. All of the FAQs for that software will apply to the **QCRTGraph** software.

1. Is the **Real-Time Graphics Tools for Java** software backward compatible with the **Charting Tools *for Windows*** and the **Graphics Class Libraries *for MFC*** ?

No, the **Real-Time Graphics Tools for Java** software is not backward compatible with earlier Quinn-Curtis products. It was developed explicitly for the new Java programming object oriented programming framework. You should have no problems recreating any charts that you created using our older Windows software; in most cases it will take far fewer lines of code. One of the few chart types that are not supported is the sweep graph.

INDEX