QE Framework -
QEGui
and
User Interface Design

Andrew Rhyder

6 January 2013

Contents

Introduction

This document describes how to use the QE Framework to develop ‘code free’ Control GUI systems. It explains how features of the QE Framework widgets can be exploited, and how the QE Framework widgets interact with each other and with the QEGui application typically used to present the user interface.

While widget properties are referenced, a definitive list of the available properties is available in document QEReferenceManual.pdf.

This document is not intended to be a general style guide, or a guide on using Qt’s user interface development tool, Designer. Style issues should be resolved using facility based style guidelines, EPICS community standards, and general user interface style guides. Consult Qt documentation regarding Designer.

Overview

In a typical configuration, Qt’s Designer is used to produce a set of Qt user interface files (.ui files) that implement an integrated GUI system. The QE Framework application QEGui is then used to present the set of .ui files to users. The set of.ui files may include custom and generic template forms, and forms can include nested sub forms. Other applications can also be integrated.

<image of a set of GUIs>

Qt Designer

Designer is used to create Qt User Interfaces containing Qt Plugin widgets. The QE Framework contains a set of Qt Plugin widgets that enable the design of Control System GUIs. These are used, along with standard Qt widgets and other third party widgets.

<image of designer>

QEGui

QEGui is an application use to display Qt User Interface files (.ui files). Almost all of the functionality of a Control System GUI based on the QE Framework is implemented by the widgets in the user interface files. QEGui simply presents these user interface files in new windows, or new tabs, and provides support such as a window menu and application wide logging.

Simple but effective integration with Qt Designer is achieved with the option of launching Designer from the QEGui ‘Edit’ Menu. The user interface being viewed can then be modified, with the changes being automatically reloaded by QEGui.

Refer to ‘QEGui’ (page 4) for documentation on using QEGui.

QE widgets

QE widgets are self contained. The application loading a user interface file – typically QEGui – does not have to be aware the user interface file even contains QE widgets. The Qt library locates the appropriate Plugin libraries that implement the widgets it finds in a user interface file.

While QE widgets need no support from the application which is loading the user interface containing them, some QE widgets are capable of interacting with the application, and other widgets. For example, a QEPushButton widget can request that whatever application has loaded it open another user interface in a new window.

QE widgets fall into two categories:

- Standard widgets. These widgets are based on a standard Qt widget and generally allow the widget to write and read data to a control system. For example, QELabel is based on QLabel and displays data updates as text.
- Control System Specific widgets. These widgets are not readily identifiable as a single standard Qt widget and implement functionality specific to Control systems. For example, QEPlot displays waveforms.

QEGui

Command format:

```
QEGui [-s] [-e] [-b] [-h] [-m.macros] [-p pathname] [filename]
```

Command switches and parameters are as follows:

- -s Single application.
  QEGui will attempt to pass all parameters to an existing instance of QEGui. When one instance of QEGui managing all QEGui windows, all windows will appear in the window menu. A typical use is when a QEGui window is started by a button in EDM.
- -e Enable edit menu option.
  When the edit menu is enabled Designer can be launched from QEGui, typically to edit the current GUI.
- -b Disable the menu bar.
- -p *path-list* Search paths. When opening a file, this list of paths may be used when searching for the file. Refer to 'File location rules' (page 3) for the rules QEGui uses when searching for a file.
- -h Display help text explaining these options.
- [-m *macros*] Macro substitutions applied to GUIs.
  Macro substitutions are in the form:
  `keyword=substitution,keyword=substitution,...` and should be enclosed

in quotes if there are any spaces.
Typically substitutions are used to specify specific variable names when loading generic template forms. Substitutions are not limited to template forms, and some QEWidgets use macro substitutions for purposes other than variable names.

- *filename* GUI filename to open.
  If no filename is supplied, the 'File Open' dialog is presented. Refer to 'File location rules' (page 3) for the rules QEGui uses when searching for a file.
- *keyword=substitution, keyword=substitution,...* Macro substitutions applied to GUIs.
  Same as –m switch, these substitutions override any –m switch.

Switches may be separate or grouped.

Switches that precede a parameter (-p, -m) may be grouped. Associated parameters are then expected in the order the switches were specified. For example:

```
QEGui -e -p /home

QEGui -epm /home PUMP=02
```

File location rules

If a user interface file path is absolute, QEGui will simply attempt to open it as is. If the file path is not absolute, QEGui looks for it in the following locations in order:

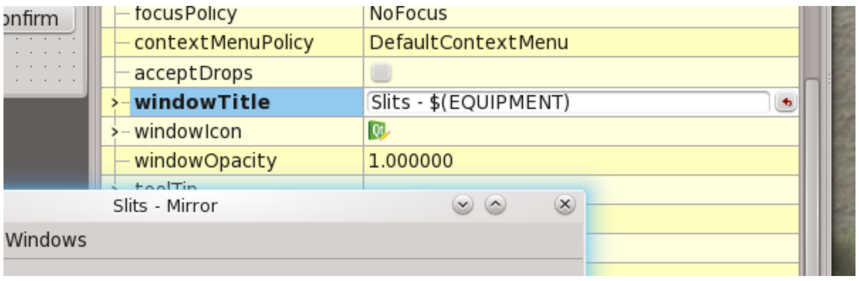
1. If the filename is for a sub-form, look in the directory of the parent form.
2. Look in the directories specified by the –p switch.
3. Look in the current directory.

<Are macro substitutions applied to filenames???>

Tricks and tips (FAQ)

GUI titles

The QEGui application reads the windowTitle property of the top level widget in a user interface file. It then applies any macro substitutions to the name and uses it as the GUI title. Figure 1 shows a windowTitle property that includes macros being edited in Designer, with the same user interface being displayed by QEGui with the appropriate macro substitution.



focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	
windowTitle	Slits - $(EQUIPMENT)
windowIcon	
windowOpacity	1.000000

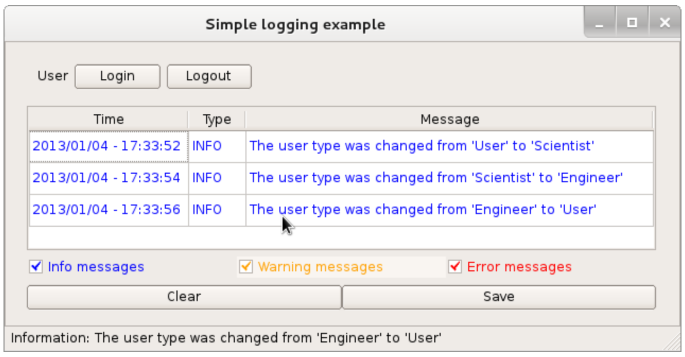
Figure 1 windowTitle Property

Logging

Several QE widgets generate log messages. These can be caught and displayed by a QELog widget, or a user application.

Simplest use:

The simplest use of this system is to drop a QELog widget onto a form. That's it. Any log messages generated by any QE widgets within the application (for example, the QEGui application) will be caught and displayed. Figure 2 shows a form containing a QELogin widget and a QELog widget. When the user logins in using the QELogin widget, messages generated by the QELogin widget are automatically logged by the QELog widget.



Time	Type	Message
2013/01/04 - 17:33:52	INFO	The user type was changed from 'User' to 'Scientist'
2013/01/04 - 17:33:54	INFO	The user type was changed from 'Scientist' to 'Engineer'
2013/01/04 - 17:33:56	INFO	The user type was changed from 'Engineer' to 'User'

Information: The user type was changed from 'Engineer' to 'User'

Figure 2 Simple logging example

Complex use:

By default, QELog widgets catch and display any massage, but messages can be filtered to display only messages from a specific sets of QE widgets or a to display messages originating from QE widgets within the same QEForm containing the QELog widget.

A form may contain QEForm widgets acting as sub forms. A QELog widget in the same form as a QEForm widget can catch and display messages from widgets in the QEForm if the QEForm is set up to catch and re-broadcast these messages. QEForm widgets can catch and filter messages exactly like QELog widgets, but selected messages are not displayed, rather they are simply re-broadcast as originating from themselves. When a QELog widget is selecting messages only from QE widgets in the same form it is in it will catch these re-broadcast messages

The messageFormFilter, messageSourceFilter, and messageSourceId properties are used to manage message filtering as follows:

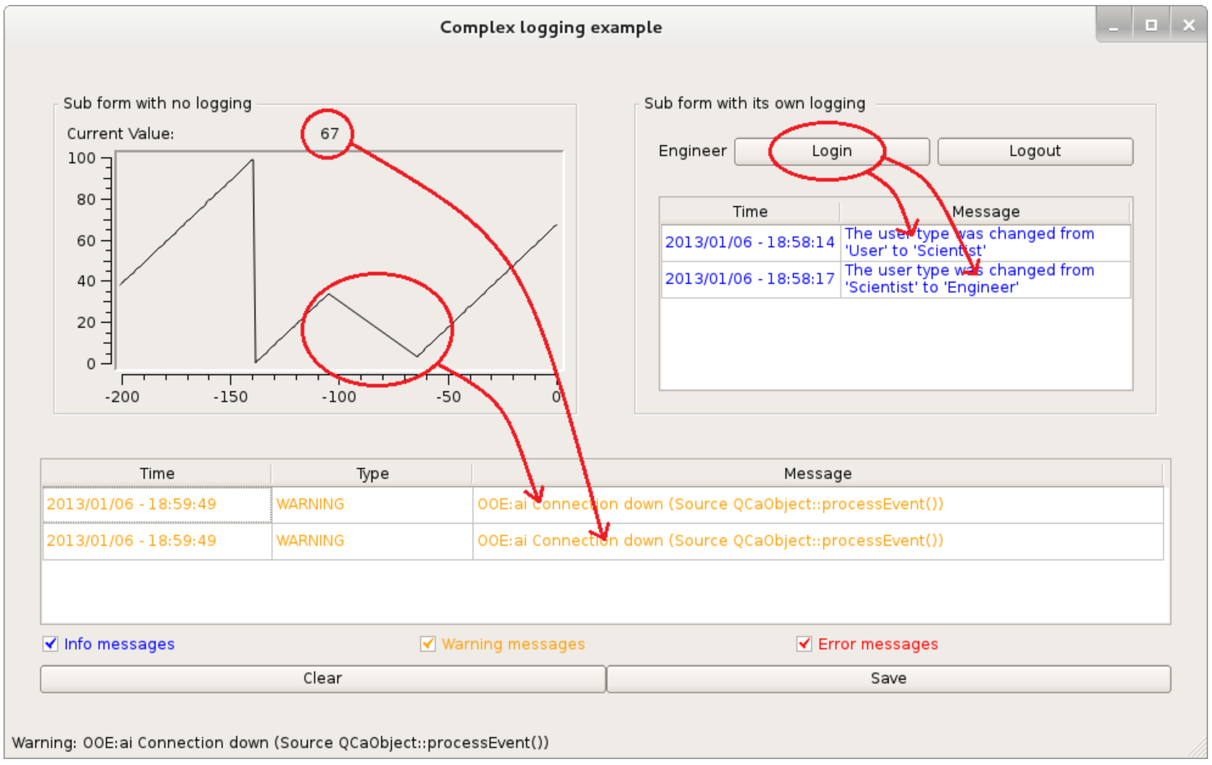
Any QE widget that generates messages has a messageSourceId property. QELog and QEForm widgets with the messageSourceId property set to the same value can then use the messageSourceFilter property to filter messages based on the message source ID as follows:

- **Any** A message will always be accepted. (messages source ID is irrelevant)
- **Match** A message will be accepted if it comes from a QEWidget with a matching message source ID.
- **None** The message will not be matched based on the message source ID. (It may still be accepted based on the message form ID.)

All generated messages are also given a message form ID. The message form ID is supplied by the QEForm the QE widget is located in (or zero if not contained within a QEForm widget). QELog and QEForm widgets with a matching message form ID can then use the messageFormFilter property to filter messages based on the message form ID as follows:

- **Any** A message will always be accepted.
- **Match** A message will be accepted if it comes from a QE widget on the same form.
- **None** The message will not be matched based on the form the message comes from. (It may still be accepted based on the message source ID.)

Figure 3 shows a complex logging example. The main form contains two sub forms and a QELog widget. The right hand sub form looks after its own messages. It has a QELog widget with filtering set to catch any messages generated on the same form. The left hand sub form does not display its own messages, but the form is set up to re-broadcast any messages generated by QE widgets it contains, so the QELog widget on the main form can be set up to catch and display these messages. Note, the QEGui application itself also uses a UserMessage class to catch and present the same messages on its status bar.



Time	Message
2013/01/06 - 18:58:14	The user type was changed from 'User' to 'Scientist'
2013/01/06 - 18:58:17	The user type was changed from 'Scientist' to 'Engineer'

Time	Type	Message
2013/01/06 - 18:59:49	WARNING	OOE:ai Connection down (Source QCaObject::processEvent())
2013/01/06 - 18:59:49	WARNING	OOE:ai Connection down (Source QCaObject::processEvent())

Warning: OOE:ai Connection down (Source QCaObject::processEvent())

Figure 3 Complex logging example

Note, Application developers can catch messages from any QE widgets in the same way the QELog and QEForm widgets do, by implementing a class based on the UserMessage class. See the UserMessage class documentation for details.

Sub form file names

Absolute names simplify locating forms, but make a set of related GUI forms and sub forms less portable. The following rules will help make a set of forms and sub forms more portable.

- No path should be specified for sub forms in the same directory as the parent form.
- A relative path should be given for sub forms in a directory under the parent form directory.
- Paths to directories containing generic sub forms can be added to the–p switch.

Refer to 'File location rules' (page 3) details on how QEGui searches for a user interface file given absolute and relative file paths.

Ensuring QERadioButton is checked if it matches the current data value

When a data update matches the checkText property, the Radio button will be checked.

If the 'format' property is set to 'Default' (which happens to be the default!), and the data has enumeration strings then the checkText property must match any enumeration string.

This can cause confusion if the values written are numerical – the click text (the value written) can end up different to the clickCheck text. Also, if the enumeration strings are dynamic, it is not possible to specify at GUI design time what enumeration strings to match.

To solve this problem, set the 'format' property to 'Integer' and set the 'checkText' property to the appropriate integer value. Remember, the checkText property is a text field that will be matched against the data formatted as text, so the checkText property must match the integer formatting. For example, a checkText property of '   2' (includes spaces) will not match '2' (no spaces)

QE widgets

QE widgets enable the design of control system user interfaces.

This document describes what the widgets are designed to do, what features they have and how they should be used. For a comprehensive list of properties, refer to the widget class documentation in QE_ReferenceManual.pdf

EPICS enabled standard Qt widgets:

Many QE widgets are simply standard Qt widgets that can generally read and write to EPICS variables. For example, a QELabel widget is basically a QLabel widget with a variable name property. When a variable name is supplied, text representing the variable is displayed in the label.

The QE Framework also manages variable status using colour, provides properties to control formatting, etc

Control System widgets

Other QE widgets implement a specific requirement of a Control System. For example QEPlot presents waveforms. These widgets are still based on standard low level Qt widgets so still benefit from common Qt widget properties for managing common properties such as geometry.

Common QE Widget properties

Properties of base Qt widgets are not documented here – refer to Qt documentation for details.

variableName and variableSubstitutions

All EPICS aware widgets have one or more variable name properties.

The variable names may contain macro substitutions that will be translated when a user interface is opened. Generally the macro substitutions will be supplied from QEGui application command line parameters, and from parent forms when a user interface is acting as a sub form. The widget itself may have default macro substitutions defined in the 'variableSubstitutions' property. Default macro substitutions are very useful when designing user interface forms as they allow live data to be viewed when designing generic user interfaces. For example, a QELabel in a generic sub form may be given the variable name SEC${SECTOR}:PMP${PUMP} and default substitutions of 'SECTOR=12 PUMP=03'. When used as a sub form valid macro substitutions will be supplied that override the default substitutions. At design time, however, the QELabel will connect to and display data for SEC12:PMP03. Note, default substitutions can be dangerous if they are never overridden.

The following example describes a scenario where macro substitutions required for a valid variable name are defined at several levels, and in one case multiple levels.

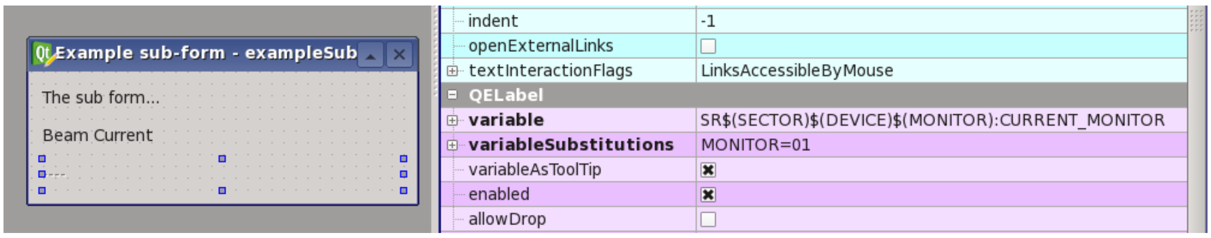
Figure 4 shows a form containing a QELabel. The variable name includes macros SECTOR, DEVICE and MONITOR. Default substitutions are provided for MONITOR. This is not adequate to derive a complete variable name.

Figure 5 shows a form using the form from Figure 4 as a sub form. Additional macro definitions for SECTOR and DEVICE are provided with the sub-form file name. When the sub form is loaded, the QELabel in the sub form can now derive a complete variable name (SR01BCM01:CURRENT_MONITOR). While complete, this is not actually functional – the correct sector is SR11.

Figure 6 shows the form from Figure 5 opened by the QEGui application with the following parameters:

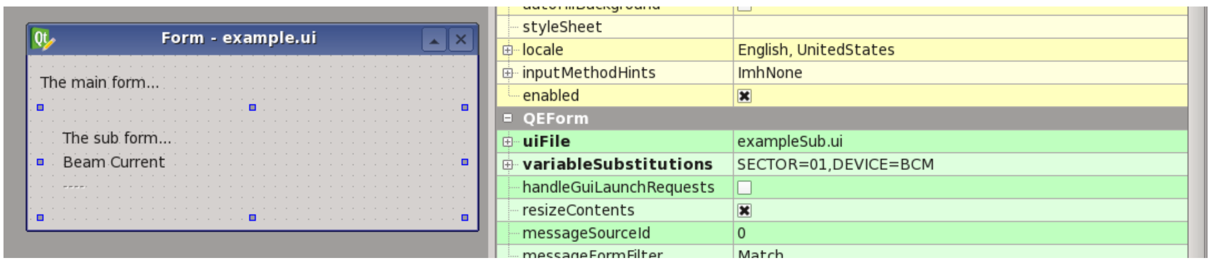
```
QEGui –m "SECTOR=11" example.ui
```

The MONITOR macro has been overwritten, so the QELabel in the sub form now derives the correct variable name SR11BCM01:CURRENT_MONITOR.



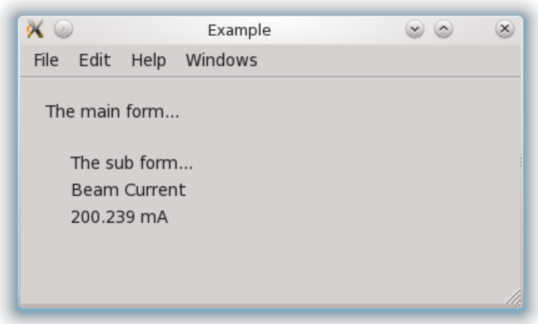
indent	-1
openExternalLinks	☐
textInteractionFlags	LinksAccessibleByMouse
QELabel	
variable	SR$(SECTOR)$(DEVICE)$(MONITOR):CURRENT_MONITOR
variableSubstitutions	MONITOR=01
variableAsToolTip	☒
enabled	☒
allowDrop	☐

Figure 4 Sub form with macro substitution for part of the variable name



styleSheet	
locale	English, UnitedStates
inputMethodHints	ImhNone
enabled	☒
QEForm	
uiFile	exampleSub.ui
variableSubstitutions	SECTOR=01,DEVICE=BCM
handleGuiLaunchRequests	☐
resizeContents	☒
messageSourceId	0
messageFormFilter	Match

Figure 5 Main form containing sub form with all macro substitutions satisfied (but one is incorrect)



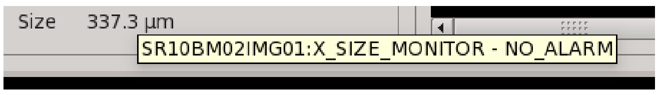
The main form...

The sub form...
Beam Current
200.239 mA

Figure 6 QEGui displaying form and sub forms with all macro substitutions satisfied correctly

variableAsTooltip

If checked, the ToolTip is generated dynamically from the variable name or names and status.

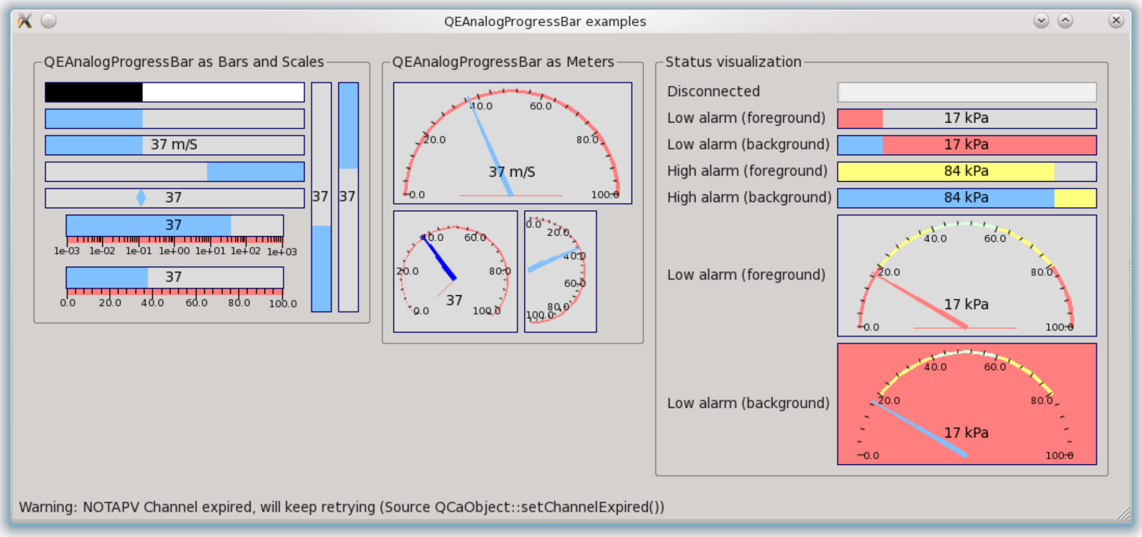


Size 337.3 µm
SR10BM02IMG01:X_SIZE_MONITOR - NO_ALARM

QEAnalogProgressBar

The QEAnalogProgressBar is used to simulate an analog indicator such as a bar indicator or dial.

- Logarithmic or linear scale
- Optional units
- Same widget used for multiple analog indicators including dial and bar.
- Based on QEAnalogIndicator which is available for non EPICS aware uses.
- Alarm Limits are represented on the scale if required

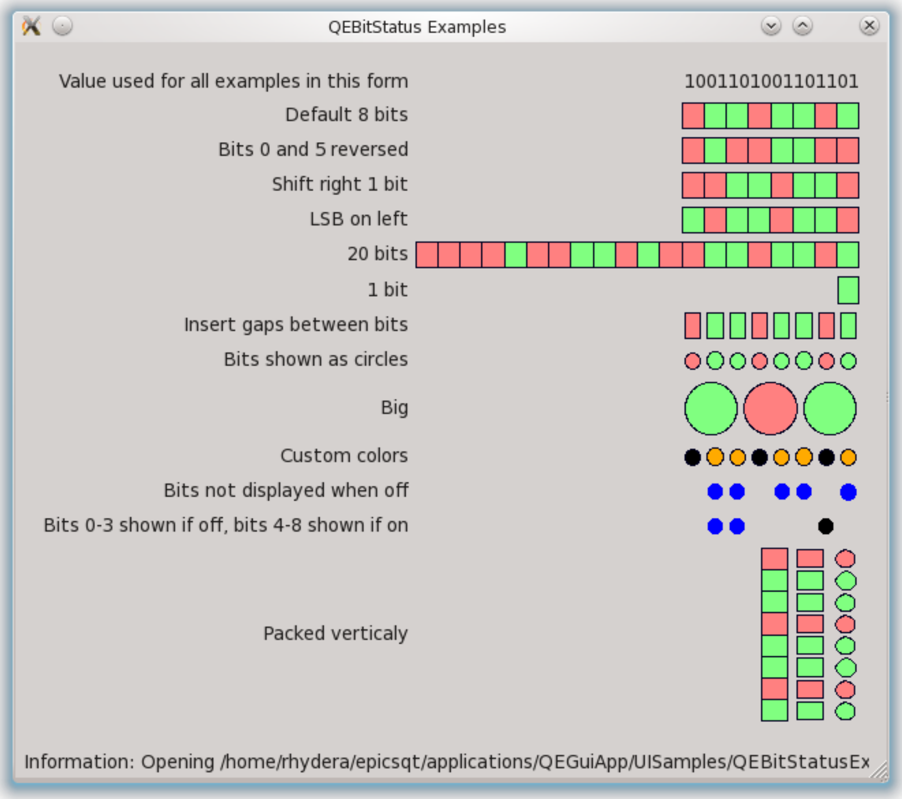


Warning: NOTAPV Channel expired, will keep retrying (Source QCaObject::setChannelExpired())

Figure 7 QEAnalogProgressBar examples

QEBitStatus

The QEBitStatus widget is used to present a selected set of bits from a data word. Bits are presented as an array of rectangles or circles with presentation properties to control shape, size, orientation, spacing and colour. Other properties allow bit by bit selection of what values display as ‘on’ and ‘off’ and if bits are rendered when ‘on’ or ‘off’.



Value used for all examples in this form 1001101001101101
Default 8 bits
Bits 0 and 5 reversed
Shift right 1 bit
LSB on left
20 bits
1 bit
Insert gaps between bits
Bits shown as circles
Big
Custom colors
Bits not displayed when off
Bits 0-3 shown if off, bits 4-8 shown if on
Packed verticaly

Information: Opening /home/rhydera/epicsqt/applications/QEGuiApp/UISamples/QEBitStatusEx

Figure 8 QEBitStatus widget examples

QEConfiguredLayout

QEFileBrowser

QELabel

QELogin

QELog

QEPvProperties

QERecipe

QEScript

QEStripChart

QEPeriodic

QESubstitutedLabel

QELineEdit

QEPushButton

QERadioButton

QEShape

QESlider

QESpinBox

QEComboBox

QEForm

QEPlot

QEImage

QEAnalogIndicator

QBitStatus

QEFrame

QEGroupBox

QELink