

QE Framework - QEGui and User Interface Design

Andrew Rhyder

8 January 2013

Contents

Introduction	4
Overview	4
Qt Designer	4
QEGui	4
QE widgets	5
QEGui	5
Command format:.....	5
File location rules	6
Tricks and tips (FAQ)	6
GUI titles	6
User levels.....	7
Logging	8
Sub form file names	10
Ensuring QERadioButton is checked if it matches the current data value	11
What top level form to use	11
QE widgets	11
Common QE Widget properties.....	12
variableName and variableSubstitutions.....	12

variableAsTooltip	13
Subscribe	13
enabled	14
allowDrop.....	14
visible	14
messageSourceId	14
userLevelUserStyle, userLevelScientistStyle, userLevelEngineerStyle.....	14
userLevelVisibility	14
userLevelEnabled	15
String formatting properties	15
precision.....	15
useDbPrecision.....	15
leadingZero	15
trailingZeros	15
addUnits.....	15
localEnumeration	16
format	17
radix	17
notation.....	17
arrayAction.....	17
arrayIndex	17
QEAnalogProgressBar	17
QEBitStatus	18
QEConfiguredLayout.....	20
QEFileBrowser.....	20
QELabel	20
QELogin	20
QELog	20
QEPvProperties	20
QERecipe	20
QEScript.....	20
QEStripChart	20

QEPeiodic.....	20
QESubstitutedLabel.....	20
QELineEdit.....	20
QEPushButton and QERadioButton	20
QERadioButton	25
QEShape.....	25
QESlider.....	25
QESpinBox.....	25
QEComboBox	25
QEForm	25
QEPlot	25
QEImage.....	25
QEAnalogIndicator	25
QBitStatus	25
QEFrame.....	25
QEGroupBox.....	25
QELink	25

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 5) 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 4) 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 4) 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.

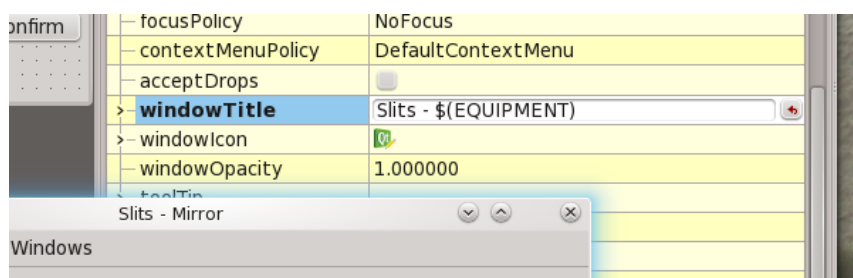


Figure 1 windowTitle Property

User levels

The QE framework manages three application wide user levels. These are independent of the operating system user accounts.

Within an application using the QE framework (such as the QEGui application), one of three user levels can be set. The three user levels are:

- **User**
- **Scientist**
- **Engineer**

User levels allow the most appropriate view of the system to be presented to different user groups. In Figure 2 for example, while in User mode operational information (beam current) is large. In Scientist mode a 'maintenance' panel appears but a maintenance control is not enabled. In Engineer mode, the maintenance control is enabled.

While the user level can be set and read programmatically using the ContainerProfile class, it is intended to be set using the QELogin widget and acted on by other QE widgets. The QELogin widget imposes a hierarchy to the user levels, requesting passwords when increasing user levels but allowing the user level to be reduced without authority.

The user levels are used to control individual QE widget behaviour. Most commonly, user level is used to determine if a QE widget is visible or enabled for a given user level through the 'userLevelVisibility' and 'userLevelEnabled' properties respectively. The 'userLevelUserStyle', 'userLevelScientistStyle' and 'userLevelEngineerStyle' properties, however, allow any style string to be applied for each user level. While user level based style strings allow many simple and convenient user interface changes beyond visibility and enabled state, they can also allow obscure and bizarre behaviour changes. For example, a style string may simply set a QEPushButton background to red in user mode, alternatively a style string could be used to move a QEPushButton to a different location on a form.

The syntax for all Style Sheet strings used by this class is the standard Qt Style Sheet syntax. For example, 'background-color: red'. Refer to Qt Style Sheets Reference for full details. The style sheet syntax includes a 'qproperty' keyword allowing any property to be altered using the style string. For

example, 'qproperty-geometry:rect(10 10 100 100);' would move a widget to position 10,10 and give it a size of 100,100.

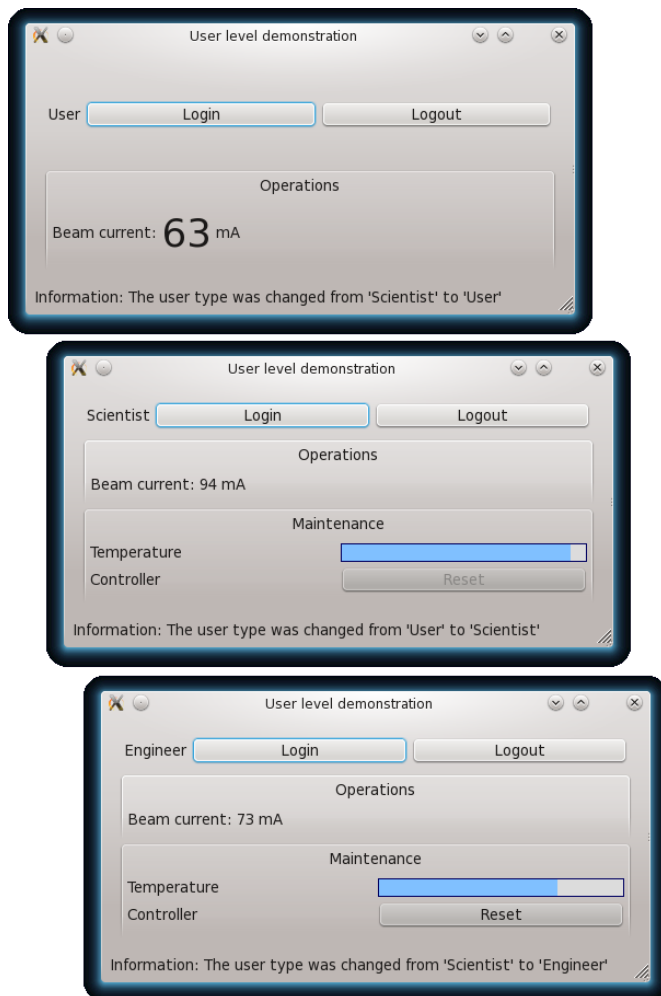


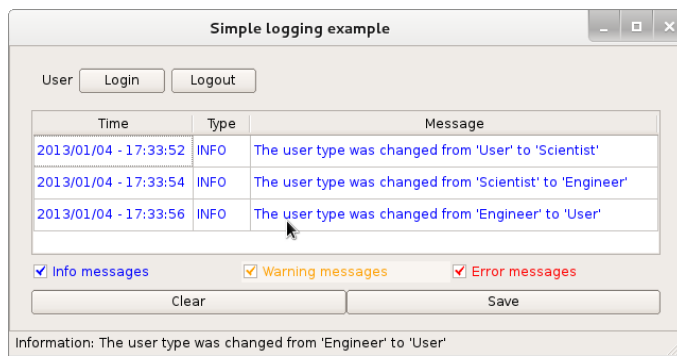
Figure 2 User level example

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 3 shows a form containing a QELogin widget and a QELog widget. When the user logs 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'

☒ Info messages
 ☐ Warning messages
 ☒ Error messages

Clear Save

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

Figure 3 Simple logging example

Complex use:

By default, QELog widgets catch and display any message, 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 QForm containing the QELog widget.

A form may contain QForm widgets acting as sub forms. A QELog widget in the same form as a QForm widget can catch and display messages from widgets in the QForm if the QForm is set up to catch and re-broadcast these messages. QForm 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 QForm 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 QForm the QE widget is located in (or zero if not contained within a QForm widget). QELog and QForm 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 4 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.

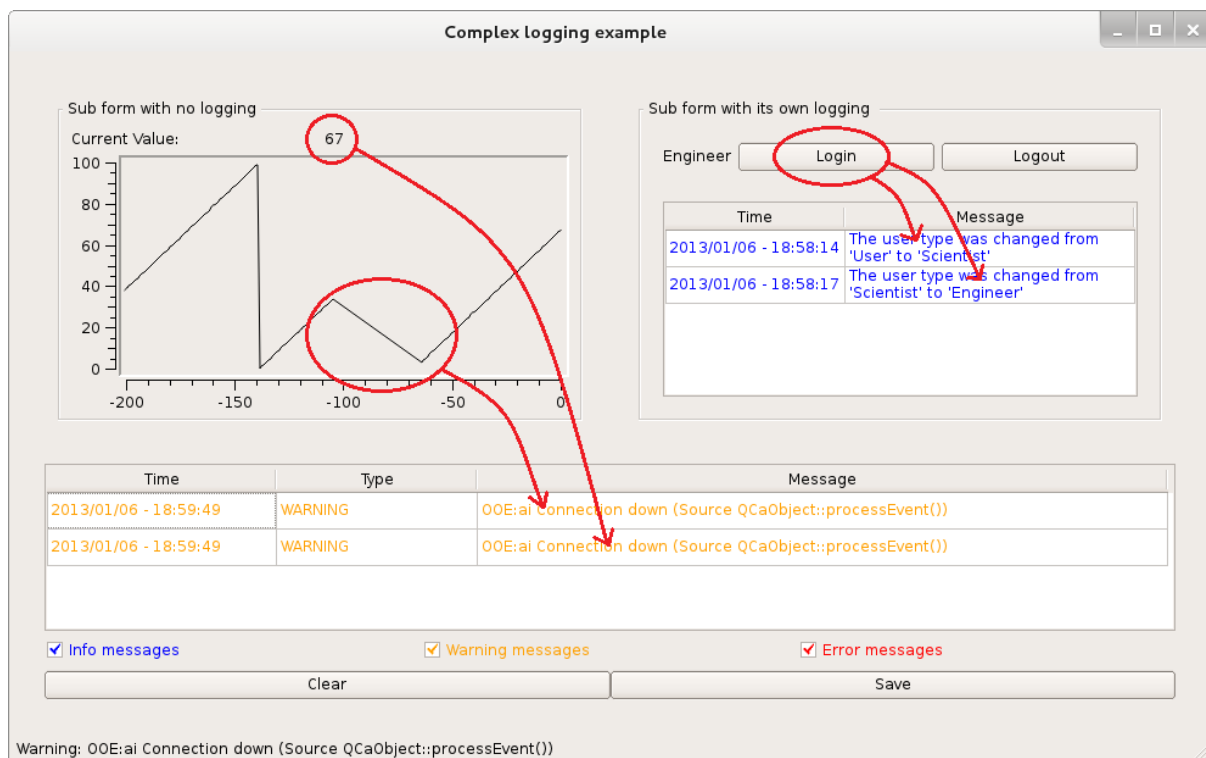


Figure 4 Complex logging example

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 4) 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)

What top level form to use

If you are using Qt's Designer to lay out user interfaces as part of an application you are developing, then the top level form you start with will depend on your application, but if you are creating a user interface file for use in QEGui the following guidelines apply:

QEGui can load a user interface file with any sort of top level widget, but the most appropriate is likely to be one of the simpler containers such as QWidget as QEGui is already managing most aspects more complex containers such as are designer to manage.

You select the top level widget when you create a new user interface in Designer. It is recommended that you choose QWidget, but if there is functionality you require provided by other widgets, then feel free to use any other widget.

QEGui opens all user interface files using a QEForm widget. If the user interface file it is opening does not have a layout, the top level widget in the user interface file is resized to match the QEForm.

If it does have a layout, then the QEForm will also have given itself a layout to ensure layout requests are propagated and the top level widget is not resized.

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 5 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 6 shows a form using the form from Figure 5 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 7 shows the form from Figure 6 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.

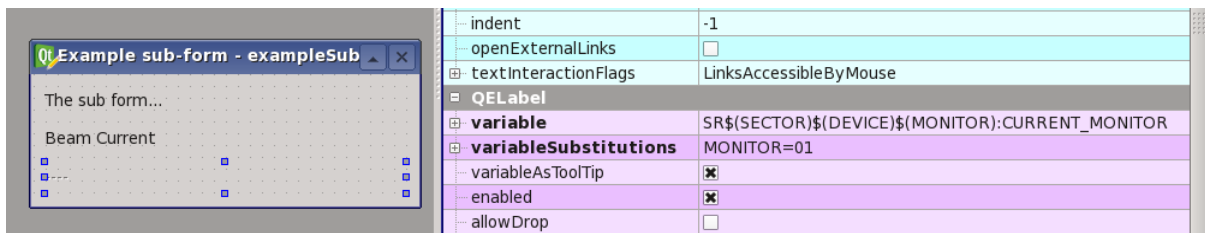


Figure 5 Sub form with macro substitution for part of the variable name

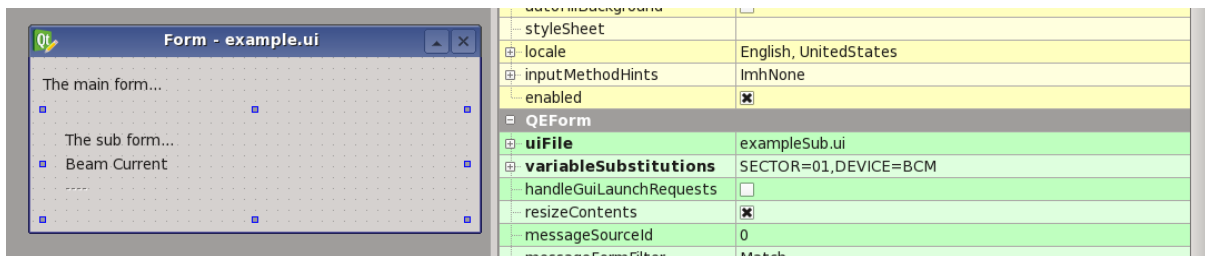


Figure 6 Main form containing sub form with all macro substitutions satisfied (but one is incorrect)

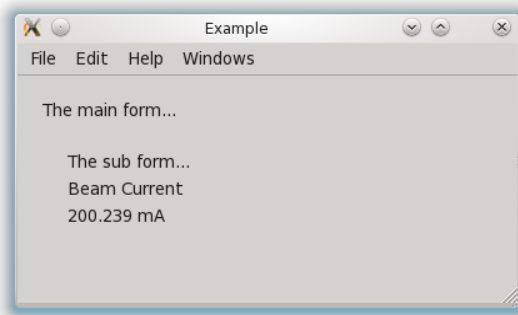
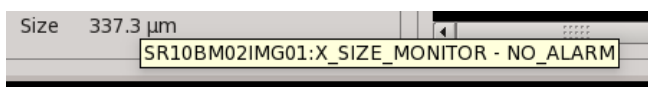


Figure 7 QEGui displaying form and sub forms with all macro substitutions satisfied correctly

variableAsTooltip

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



Subscribe

enabled

Set the preferred 'enabled' state. Default is true.

The standard Qt 'enabled' property is set false by many QE widgets to indicate if the data displayed is invalid (disabled). When the data displayed is valid, the QE widget will reset standard Qt 'enabled' property to the value of this 'enabled' property. Users wanting to enable or disable a QE widget for other purposes should use this property. This property will be used to set the standard Qt 'enabled' property except when data is invalid.

allowDrop

Allow drag/drops operations to this widget. Default is false. Any dropped text will be used as a new variable name.

visible

Display the widget. Default is true. Setting this property false is useful if widget is only used to provide a signal - for example, when supplying data to a QELink widget.

Note, when false the widget will still be visible in Qt Designer.

messageSourceId

Set the ID used by the message filtering system. Default is zero.

Widgets or applications that use messages from the framework have the option of filtering on this ID.

For example, by using a unique message source ID a QELog widget may be set up to only log messages from a select set of widgets.

Refer to Logging (page 8) for further details.

userLevelUserStyle, userLevelScientistStyle, userLevelEngineerStyle

Style Sheet strings to be applied when the widget is displayed in 'User', 'Scientist', or 'Engineer' mode. Default is an empty string.

The syntax is the standard Qt Style Sheet syntax. For example, 'background-color: red'

This style strings will be safely merged with any existing style string supplied by the application environment for this widget, or any style string generated for the presentation of data.

Refer to User levels (page 7) for details regarding user levels.

userLevelVisibility

Lowest user level at which the widget is visible. Default is 'User'.

Used when designing GUIs that display more detail according to the user mode.

The user mode is set application wide through the QELogin widget, or programatically through `setUserLevel()`.

Widgets that are always visible should be visible at 'User'.

Widgets that are only used by scientists managing the facility should be visible at 'Scientist'.

Widgets that are only used by engineers maintaining the facility should be visible at 'Engineer'.

Refer to User levels (page 7) for details regarding user levels.

userLevelEnabled

Lowest user level at which the widget is enabled. Default is 'User'.

Used when designing GUIs that allow access to more detail according to the user mode.

The user mode is set application wide through the QELogin widget, or programatically through `setUserLevel()`

Widgets that are always accessible should be visible at 'User'.

Widgets that are only accessible to scientists managing the facility should be visible at 'Scientist'.

Widgets that are only accessible to engineers maintaining the facility should be visible at 'Engineer'.

Refer to User levels (page 7) for details regarding user levels.

String formatting properties

Many QE widgets present data as text, or interpret text and write data accordingly. Examples, are QELabel and QELineEdit.

Common formatting properties are used for all these widgets where possible. Not all are relevant for all data types.

precision

Precision used when formatting floating point numbers. The default is 4.

This is only used if the 'useDbPrecision' property is false.

useDbPrecision

If true (default), format floating point numbers using the precision supplied with the data.

If false, the 'precision' property is used.

leadingZero

If true (default), always add a leading zero when formatting numbers.

trailingZeros

If true (default), always remove any trailing zeros when formatting numbers.

addUnits

If true (default), add engineering units supplied with the data.

localEnumeration

An enumeration list used to data values. Used only when the 'format' property set to 'local enumeration'.

The data value is converted to an integer which is used to select a string from this list.

Format is:

```
[[<|<=|=|!=|>|=|>]value1|*] : string1 , [[<|<=|=|!=|>|=|>]value2|*] : string2 , [[<|<=|=|!=|>|=|>]value3|*] : string3 , ...
```

Where:

- < Less than
- <= Less than or equal
- = Equal (default if no operator specified)
- >= Greather than or equal
- > Greater than
- * Always match (used to specify default text)

Rules are:

- Values may be numeric or textual
- Values do not have to be in any order, but first match wins
- Values may be quoted
- Strings may be quoted
- Consecutive values do not have to be present.
- Operator is assumed to be equality if not present.
- White space is ignored except within quoted strings.
- \n may be included in a string to indicate a line break

Examples:

- 0:Off,1:On
- 0 : "Pump Running", 1 : "Pump not running"
- 0:"" , 1:"Warning!\nAlarm"
- <2:"Value is less than two", =2:"Value is equal to two", >2:"Value is grater than 2"
- 3:"Beamline Available", *:""

- "Pump Off": "OH NO!, the pump is OFF!", "Pump On": "It's OK, the pump is on"

The data value is converted to a string if no enumeration for that value is available.

For example, if the local enumeration is '0:off,1:on', and a value of 10 is processed, the text generated is '10'.

If a blank string is required, this should be explicit. for example, '0:off,1:on,10: ""'

A range of numbers can be covered by a pair of values as in the following example:

- >=4: "Between 4 and 8", <=8: "Between 4 and 8"

format

This property indicates how non textual data is to be converted to text:

- Default Format as best appropriate for the data type.
- Floating Format as a floating point number
- Integer Format as an integer
- UnsignedInteger Format as an unsigned integer
- Time Format as a time
- LocalEnumeration Format as a selection from the 'localEnumeration' property

radix

Base used for when formatting integers. Default is 10 (duh!)

notation

Notation to use when formatting data as a floating point number. Default is Fixed. Options are:

- Fixed Standard floating point. For example: 123456.789
- Scientific Scientific representation. For example: 1.23456789e6
- Automatic Automatic choice of standard or scientific notation

arrayAction

This property defines how array data is formatted as text. Default is ASCII. Options are:

- **Append** Interpret each element in the array as an unsigned integer and append string representations of each element from the array with a space in between each. For example, an array of three numbers 10, 11 and 12 will be formatted as '10 11 12'.
- **Ascii** Interpret each element from the array as a character in a string. Translate all non printing characters to '?' except for trailing zeros (ignore them). For example an array of three characters 'a' 'b' 'c' will be formatted as 'abc'.
- **Index** Interpret the element selected by setArrayIndex() as an unsigned integer. For example, if arrayIndex property is 1, an array of three numbers 10, 11 and 12 will be formatted as '11'.

arrayIndex

Index used to select a single item of data for formatting from an array of data. Default is 0.

Only used when the arrayAction property is INDEX. Refer to the 'arrayAction' property for more details.

QEAnalogProgressBar

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

Features include:

- 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

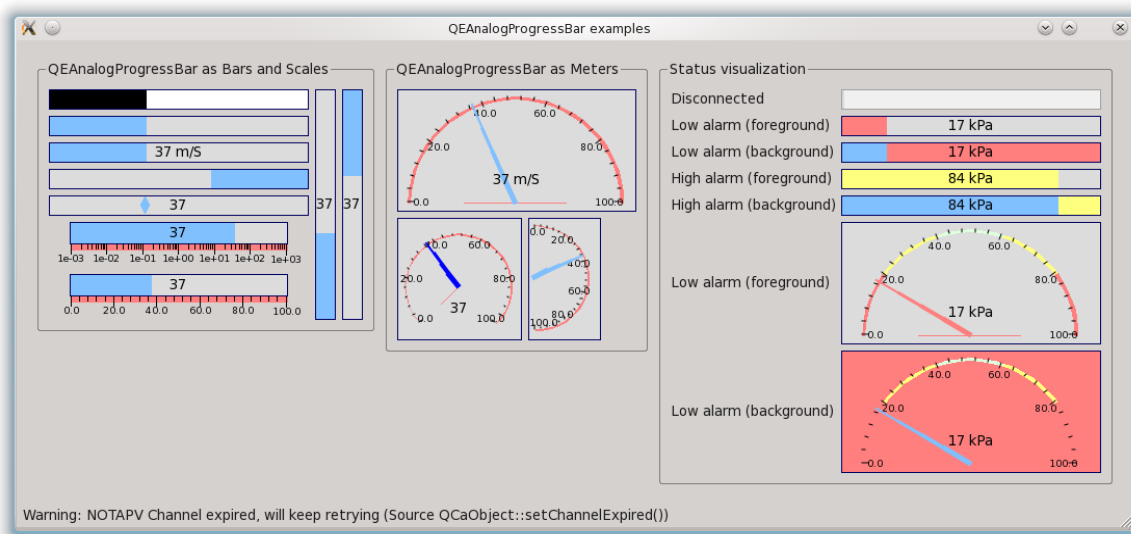


Figure 8 QEAnalogProgressBar examples

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'.

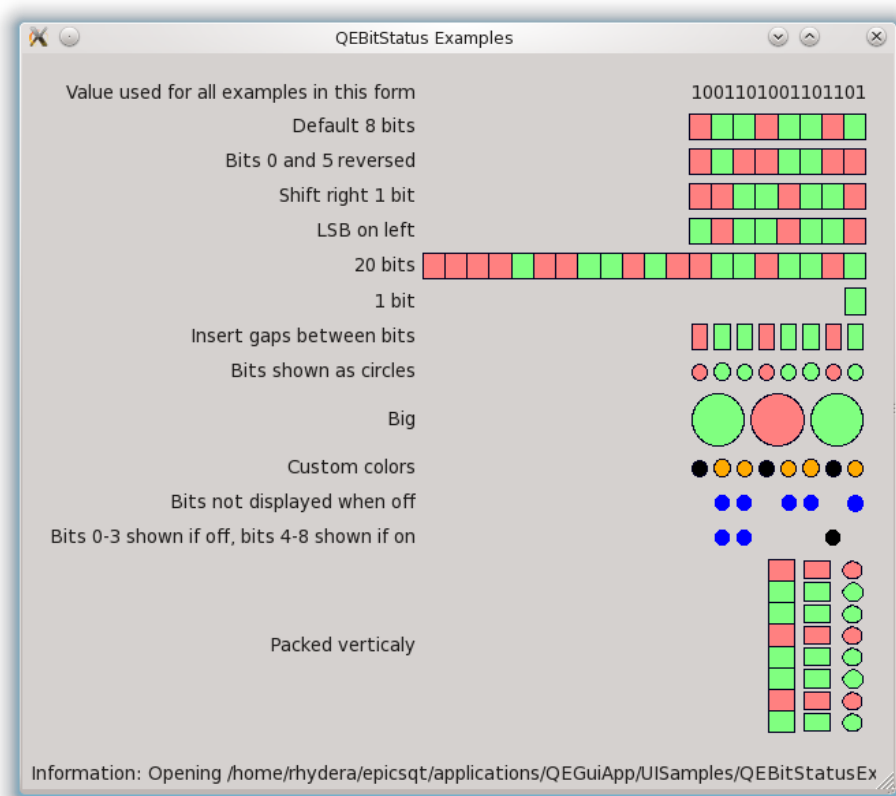


Figure 9 QEBitStatus widget examples

[QEConfiguredLayout](#)

[QEFileBrowser](#)

[QELabel](#)

[QELogin](#)

[QELog](#)

[QEPvProperties](#)

[QERecipe](#)

[QEScript](#)

[QEStripChart](#)

[QEPeriodic](#)

[QESubstitutedLabel](#)

[QELineEdit](#)

[QEPushButton and QERadioButton](#)

The QEPushButton and QERadioButton widgets provide the following non exclusive functions:

- Write to a variable
- Read from a variable
- Issue a command to the operating system
- Open a new GUI form.

If the properties used to define any or all of these functions are set up, the function will be carried out.

Both widget types are based on QEGenericButton and on QAbstractButton (through QPushButton and QRadioButton). QEPushButton and QERadioButton widgets share most properties and it is mainly the way the buttons are presented that differentiates them.

Generally, QERadioButtons will be shown as checkable, and properties related to the checked state are more likely to be used for QERadioButtons.

Various data values can be written on any or all or the following button actions:

- Press A mouse press with the pointer over the button

- **Release** A mouse release with the pointer over the button
- **Click** A press and release while over the button

By default, values are written on a button click. A click will be accompanied with a press and release.

Writing values on Press and Release typically allows a value to be set momentary, while the button is held down. In this case, no data would be written on the click.

While QPushButton and QRadioButton widgets can open a new GUI form when set up correctly without any action on the part of the application that created them, this functionality is mainly so the button functionality can be tested from the Designer 'preview' window. Applications using QPushButton and QRadioButton widgets should provide a slot to create new windows through the ContainerProfile class. The application can then respect the creation options set up with the new button and manage the window better – for example it may wish to add the window to its window menu. The QEGui application provides such a slot through the ContainerProfile class. Refer to the QEGui application and the Container Profile class for more details.

To write to a variable, the following properties are used:

- **variable**
If present, a value will be written to the variable when the button is operated.
The value of this variable can also be used to update the button text or image.
- **variable Substitutions**
Macro substitutions to apply to 'variable' and 'altReadbackVariable' properties
- **password**
Password user will need to enter before any action is taken.
- **confirmAction**
If true, a dialog will be presented asking the user to confirm if the button action should be carried out
- **writeOnPress**
If true, the 'pressText' property is written when the button is pressed. Default is false
- **writeOnRelease**
If true, the 'releaseText' property is written when the button is released. Default is false
- **writeOnClick**
If true, the 'clickText' property is written when the button is clicked. Default is true
- **pressText**
Value written when user presses button if 'writeOnPress' property is true
- **releaseText**
Value written when user releases button if 'writeOnRelease' property is true
- **clickText**
Value written when user clicks button if 'writeOnClick' property is true

- **clickCheckedText**

Text used to compare with text written or read to determine if push button should be marked as checked.

Note, must be an exact match following formatting of data updates.

When writing values, the 'pressText', 'ReleaseText', or 'clickedtext' must match this property to cause the button to be checked when the write occurs.

- **Good example:** formatting set to display a data value of '1' as 'On', clickCheckedText is 'On', clickText is 'On'. In this example, the push button will be checked when a data update occurs with a value of 1 or when the button is clicked.
- **Bad example:** formatting set to display a data value of '1' as 'On', clickCheckedText is 'On', clickText is '1'. In this example, the push button will be checked when a data update occurs with a value of 1 but, although a valid value will be written when clicked, the button will not be checked when clicked as '1' is not the same as 'On'.

To read from a variable, the following properties are used:

- **subscribe**

If checked, the button will read and present the current value defined by the 'variable' property. If the 'altReadbackVariable'.property is define, it is used in preference to the 'variable' property

- **variable**

If present, a value will be written to the variable when the button is operated. The value of this variable can also be used to update the button text or image.

- **altReadbackVariable**

If present, the value of this variable will be used to update the button text or image if required.

- **variable Substitutions**

Macro substitutions to apply to 'variable' and 'altReadbackVariable' properties

- **updateOption**

Used to determine if the data is presented textually using the button's 'text' property, or graphically using the button's 'icon' property, both textually and graphically, or if the data updates the buttons checked state.

Options are:

- Text Data updates will update the button text
- Icon Data updates will update the button icon
- TextAndIcon Data updates will update the button text and icon
- State Data updates will update the button state (checked or unchecked)

- **Pixmap0 to pixmap7**

Pixmap to display if updateOption is Icon or TextAndIcon and data value translates to an index between 0 and 7.

- **alignment**

Set the buttons text alignment.

Left justification is particularly useful when displaying quickly changing numeric data updates.

General presentation:

- **labelText**

Button label text (prior to substitution).

Macro substitutions will be applied to this text and the result will be set as the button text.

Used when data updates are not being represented in the button text.

For example, a button in a sub form may have a 'labelText' property of 'Turn Pump \$(PUMPNUM) On'.

When the sub form is used twice in a main form with substitutions PUMPNUM=1 and PUMPNUM=2 respectively, the two identical buttons in the sub forms will have the labels 'Turn Pump 1 On' and 'Turn Pump 2 On' respectively.

A system command can be issued on a button click using the following properties:

- **program**

Program to run when the button is clicked.

No attempt to run a program is made if this property is empty.

Example:

```
firefox
```

- **arguments**

Arguments for program specified in the 'program' property.

A new GUI can be started on a button click using the following properties:

- **guiFile**

File name of GUI to be presented on button click.

File name can be absolute, relative to the path of the QEform in which the QEPushButton is located, relative to the any path in the path list published in the ContainerProfile class, or relative to the current path.

See QEWidget::openQEFile() in QEWidget.cpp for details on the rules used for locating a user interface file.

- **creationOption**

Creation options when opening a new GUI. Open a new window, open a new tab, or replace the current window.

The creation option is supplied when the button generates a newGui signal.

Application code connected to this signal should honour this request if possible.

When used within the QEGui application, the QEGui application creates a new window, new

tab, or replaces the current window as appropriate.

Options are:

- Open Replace the current GUI with the new GUI
- NewTab Open new GUI in a new tab
- NewWindow Open new GUI in a new window



Figure 10 QEPushButton and QERadioButton examples

QERadioButton

QEShape

QESlider

QESpinBox

QEComboBox

QEForm

QEPlot

QEImage

QEAnalogIndicator

QBitStatus

QEFrame

QEGroupBox

QELink