EPICS development using Qt

Andrew Rhyder

Senior Controls Engineer

Australian Synchrotron

15th April 2012

Training package

The examples throughout this document are available on a Fedora 16 virtual machine which can be installed as follows from the supplied media:

- Install Oracle VirtualBox 4.1.12
- Install Oracle VirtualBox 4.1.12 extension pack
- Start VirtualBox and import appliance Fedora16.ova
- The supplied Fedora appliance is set to use 1GB of the host system memory. If the host computer cannot spare this, reduce the memory usage in the VirtualBox System settings for the Fedora16 appliance.
- Start the Fedora virtual machine
- Log in as Andrew (password andrew)
- Root password if required is Fedora16

Qt past present and future

Very brief history

- 1995 First release - Quasar Technologies / Troll Tech / Trolltech
- 1999 Qt 2.0
- 2001 Qt 3.0
- 2005 Qt 4.0
- 2008 Nokia acquired Trolltech with plans to make Qt the main development platform for its devices, including Symbian devices
- Feb 2011 Nokia drops Symbian
- March 2011 Digia acquire Qt Commercial software licensing business from Nokia
- Oct 2011 Qt Project goes live

Licencing

Qt has always had some form of 'open' licence. GPL in 2005 and LGPL in 2009

User base

5000 Commercial customers and 500,000 open source developers.

Downloads:

- 2007 300,000
- 2010 1,500,000

Products:

- Adobe Photoshop Elements
- Skype
- VLC media player
- VirtualBox

Ownership

Nokia owns Qt and still provides the main development effort for it.

It is now developed as a an open source project, the Qt Project, involving both individual developers as well as developers from Nokia

The future

- 2012 Qt 5
- A gradual shift in GUI development away from the toolkit's C++ roots to 'a declarative scripting framework' - QML and JavaScript. The heavy lifting will still be done with C++, with JavaScript as the glue to handle user interface events.
- The framework has moved to open governance with ongoing development of Qt 5 being done at qt-project.org. It is now possible to submit and review patch for developers outside Nokia.
- Around 250 developers working on Qt

More than GUIs – Qt is a complete C++ framework

GUI stuff:

- Widgets
- Event handling
- Drag and drop
- Internationalisation
- 2D and 3D graphics

Non-GUI stuff:

- Signals and slots
- Container classes
- IO
- Databases
- Multithreading
- Networking
- XML

User Interface definitions

Qt User Interfaces can be defined during application development either in your code, or graphically using Qt's Designer.

User interfaces can also be loaded at run time. This feature enables the QCa framework GUI application ASgui to rely completely on Qt to load GUI files.

Defining user interfaces in code

I have had very little need to do this. In fact after a quick browse through the QCa framework code, I can only find a few places in the ASgui application where I manipulate the higher level windows, and an example where I build up a very repetitive form based on the periodic table:

In all cases I am adding widgets that contain a hierarchy of widgets defined in Qt's Designer. Building a widget hierarchy completely in code is rarely nessesary.

Adding a GUI as a new tab: MainWindow.cpp, MainWindow::loadGuiIntoCurrentWindow()

```
    QTabWidget* tabs = getCentralTabs();
    if( tabs )
    {
        int i = tabs->currentIndex();
        tabs->removeTab( i );
        tabs->insertTab( i, gui, gui->getASGuiTitle() );
        tabs->setCurrentWidget( gui );
    }
```

Replacing the GUI in the main window: MainWindow.cpp, MainWindow::loadGuiIntoCurrentWindow()

```
    setCentralWidget( gui );
```

Adding a small for for each element in a periodic table: PeriodicSetupDialog .cpp PeriodicSetupDialog::PeriodicSetupDialog()

```
for( int i = 0; i < NUM_ELEMENTS; i++ )
{
    elements[i] = new PeriodicElementSetupForm( this );
    ...
    ...
    periodicGrid->addWidget( elements[i],
                             QCaPeriodic::elementInfo[i].tableRow,
                             QCaPeriodic::elementInfo[i].tableCol );
}
```

Manipulating a user interface through code is the best choice when the application needs to dynamically modify the user interface. For example, a large proportion of the widgets are created or deleted by the application at run time. I can't think of many user friendly forms that would do this.

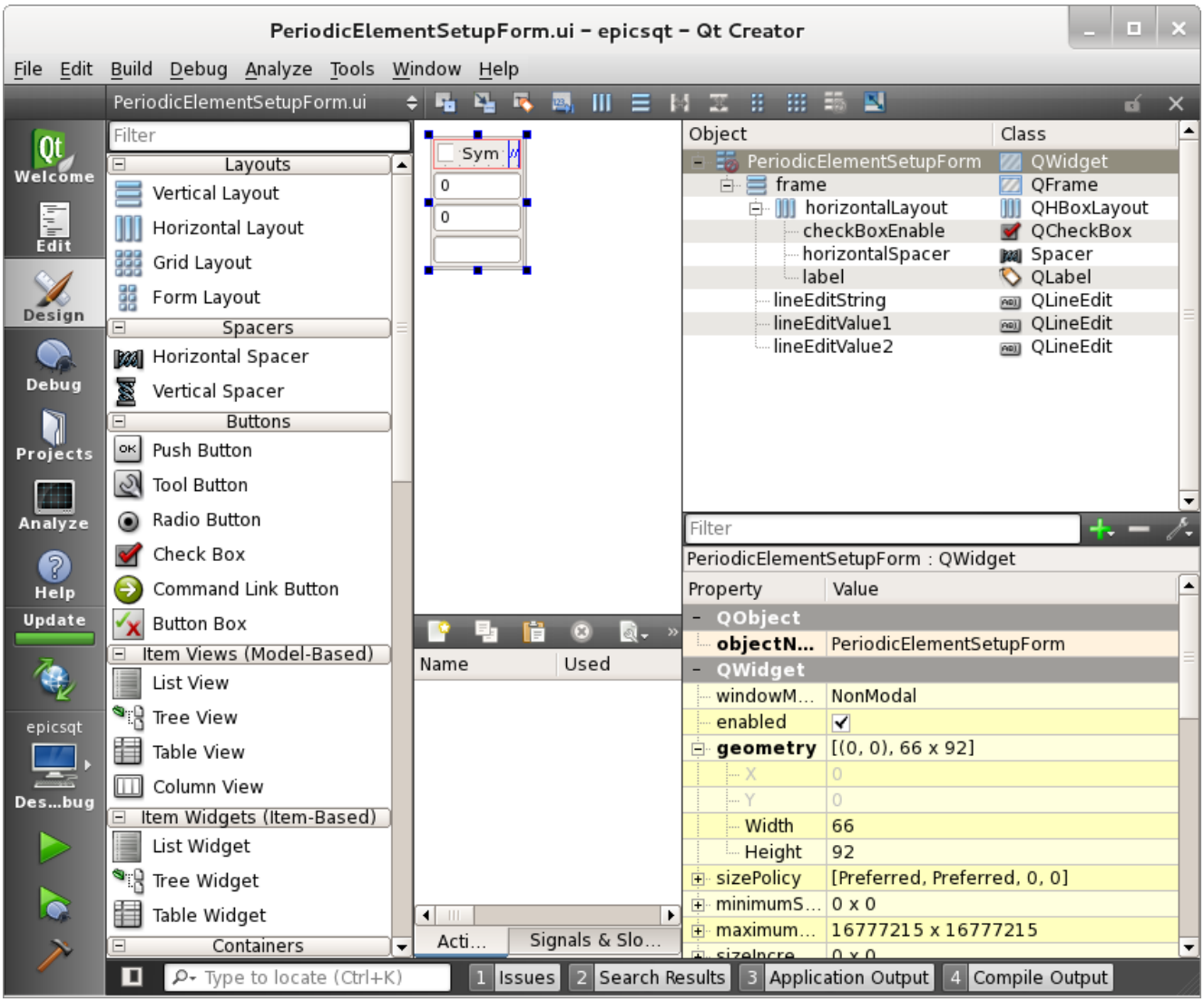
Defining user interfaces in Qt's designer

Designer is now very well integrated within QtCreator, but can still be used as a standalone form editor.

Designer creates XML User Interface definitions, with a .ui filename extension, that can be used as source files in a Qt project, or loaded at run time by Qt.

When used as source code, Qt's meta object compiler generates C++ from the XML definitions that are then included with the project C++ source.

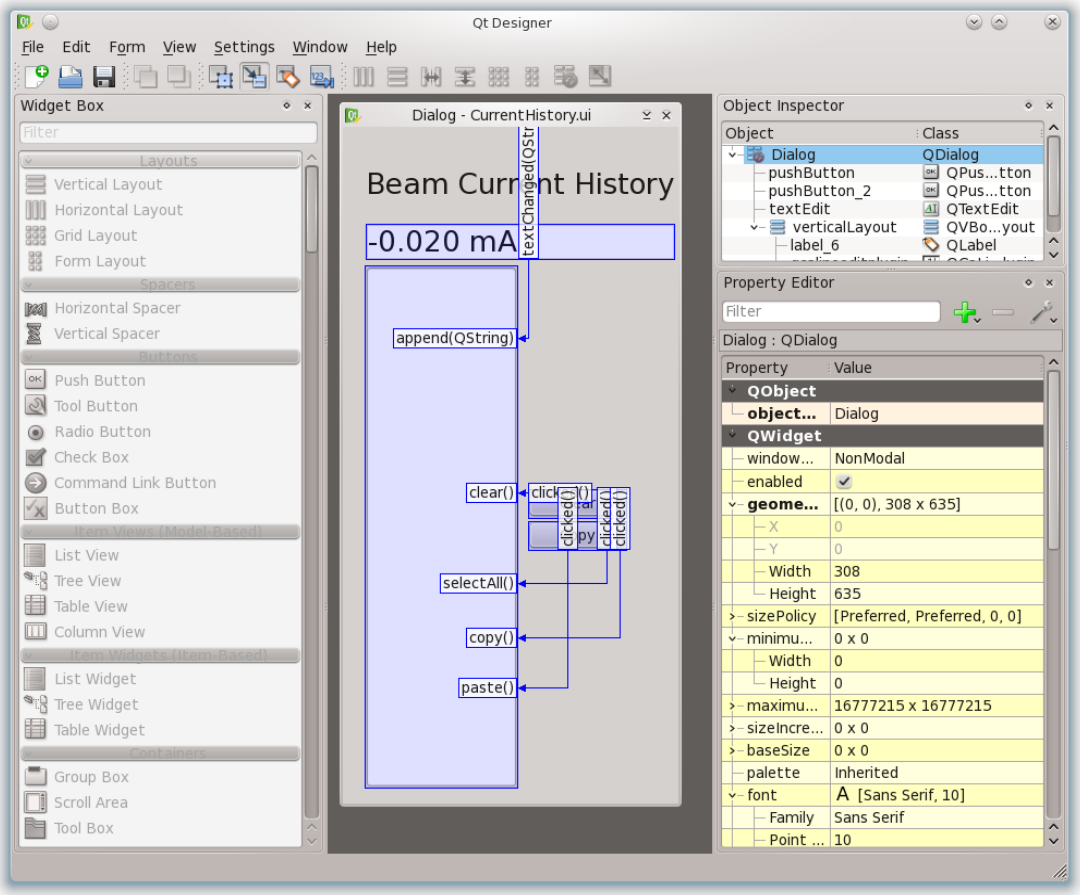
For example the QCa framework includes a user interface definition PeriodicElementSetupForm.ui, shown here in Designer (within QtCreator):

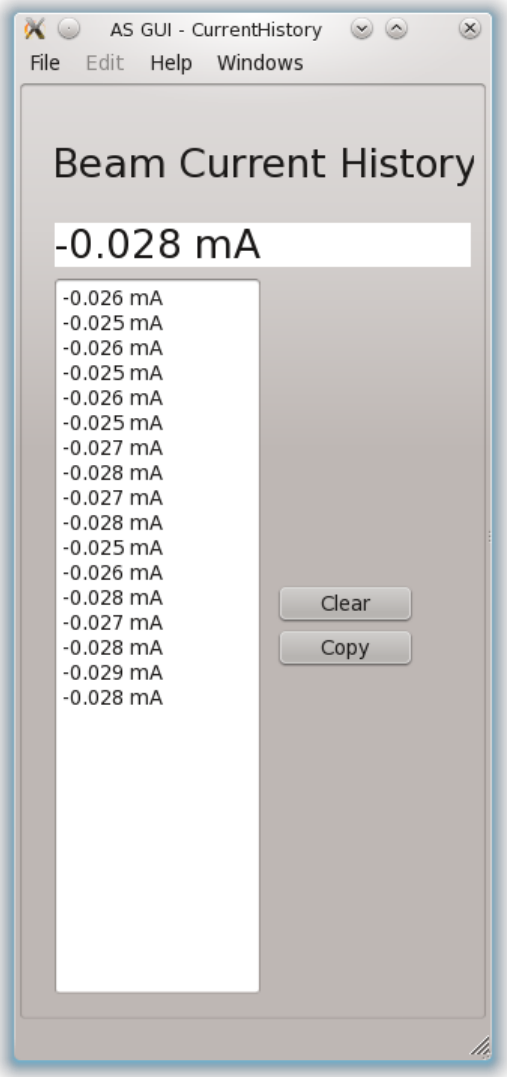


The meta object compiler generates moc_PeriodicElementSetupForm.cpp (in the build directory) from PeriodicElementSetupForm.ui.

The ability to load .ui files at run time is the reason the QCa framework's GUI application can be as light as it is. It does no more than provide an appropriate empty window and request Qt loads a .ui file into it.

Designer also allows the graphical definition of signal/slot connections between widgets. For example, in the following control system GUI, the beam current label emits a 'text changed' signal that is connected to the 'set text' slot of the current history list widget. Push button signals are also connected to the list widget to clear it, and copy it's contents to the clipboard. These signal/slot connections were defined in Designer when the GUI was created. When the user interface file is loaded by ASgui, this signal/slot connection is created without any action on the part of the ASgui application.

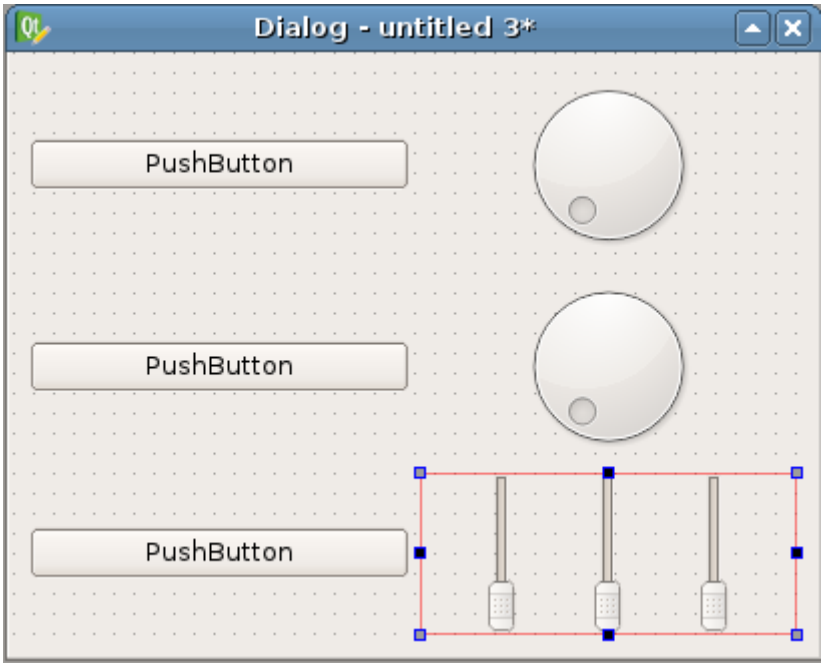

Qt Designer
File Edit Form View Settings Window Help
Widget Box
Layouts
Vertical Layout
Horizontal Layout
Grid Layout
Form Layout
Spacers
Horizontal Spacer
Vertical Spacer
Buttons
Push Button
Tool Button
Radio Button
Check Box
Command Link Button
Button Box
Item Views (Model-Based)
List View
Tree View
Table View
Column View
Item Widgets (Item-Based)
List Widget
Tree Widget
Table Widget
Containers
Group Box
Scroll Area
Tool Box
Dialog - CurrentHistory.ui
Beam Current History
-0.020 mA
textChanged(QStr
append(QString)
clear()
selectAll()
copy()
paste()
Object Inspector
Object
Class
Dialog
QDialog
pushButton
QPus...tton
pushButton_2
QPus...tton
textEdit
QTextEdit
verticalLayout
QVBo...yout
label_6
QLabel
Property Editor
Filter
Dialog : QDialog
Property
Value
QObject
object...
Dialog
QWidget
window...
NonModal
enabled
geome...
[(0, 0), 308 x 635]
X
0
Y
0
Width
308
Height
635
sizePolicy
[Preferred, Preferred, 0, 0]
minimu...
0 x 0
Width
0
Height
0
maximu...
16777215 x 16777215
sizeIncre...
0 x 0
baseSize
0 x 0
palette
Inherited
font
A [Sans Serif, 10]
Family
Sans Serif
Point ...
10



The concepts of creating widgets in code or using a form designer to aid user interface design are not new. X/Motif developers who have used one of the many forms designers have always had both options. Three differences in concept stand out:

- By allowing the definition of Signals and Slot connections by Designer, as mentioned earlier, Designer can embed application functionality into the user interface definition. This concept of defining more than just the widget layout in the form designer is set to grow with Qt 5, with QML and JavaScript used to embed a lot more application logic in the user interface definition. Or so I hear.
- The output of the form designer can be used at run time.
- The layout is managed in a much more prescribed way.
  In Motif, the developer can define attachments between widgets that manage their interrelationships. These attachments determine how widget geometry is managed when a window is resized for example. Motif guidelines are provided, but the attachment mechanism is very general and the developer is free to do whatever they like.
  In Qt, a layout widget is used to impose one of a set of behaviour models. For example, layout the widgets in a grid. While the developer is not free to do whatever they like, it inforces consistency and

there are many layout properties to allow required customisation, such as setting column proportions in a grid layout. Also, layouts can be nested. For example, the following form has a grid layout, but a horizontal layout containing three sliders has been added in the lower right cell.



Plugins

A Qt plugin is a shared library with a defined interface. The interface can be one of 15 or so standard Qt plugin types and are typically used to add database drivers, image formats, styles, etc to Qt. When Qts Designer is used within QtCreator, it is loaded as a Qt Plugin.

Here we will concentrate on generating plugin libraries to extend the widgets available when creating User Interface definitions in Designer.

We can populate such a plugin library with EPICS CA aware widgets that can be used within Designer and with applications that use Qt to load User Interface definitions created by Designer.

ASgui is such an application.

ASgui uses the QLoader class to load a .ui file created by designer. QLoader is no doubt how Designer itself loads .ui files. QLoader looks for plugin libraries, parses the .ui file, and uses the plugin libraries to generate the user interface.

A side track: The ASguiForm widget allows nested user interfaces in GUIs. When a user interface is created using QLoader, and one of the widgets in the user interface is a ASguiForm, then that ASguiForm will create a QScrollArea widget and populate it with a sub form. It will achieve this by itself calling QLoader with the sub form .ui file name. The sub form can itself have a sub form and so on.

The ASguiForm is not just for sub forms, however. The ASgui application uses an ASguiForm widget in each of its main windows, since the ASguiForm widget already knows how to load .ui files and manage a few other important tasks such as distributing EPICS macro substitutions to interested widgets.

Back on track: While the ASgui application loads the QCaPlugin library directly so it can create QAguiForm widgets, QLoader is unaware of this and does its normal search for plugin libraries when creating widgets, and so will dynamically load the QCaPlugin library for all QCa widget creation.

Building

Qt macros and Qt interface definition classes are used heavily to build a shared library with the functions required to enable the library to conform to the appropriate Qt plugin interface definition.

For a widget to be useable in Designer it must use the Q_PROPERTY macro to define properties, and the library must contain a manager class that can, among other tasks, to create the widget. The manager class must be based on QDesignerCustomWidgetInterface.

The QCa widgets are generally written as a conventional widget with no plugin functionality. A wrapper class is then added to add plugin properties, and manager class added to allow the library to generate the widgets as required.

For example, the QCa framework includes the following to implement a CA aware label widget within a plugin library that can be used by Designer and QLoader:

- QCaLabel class.
  This is based on QLabel and allows CA data to be displayed in a label. There is nothing in QCaLabel that is specific to Qt plugins.
  QCaLabel.h:

```
class QCAPLUGINLIBRARYSHARED_EXPORT QCaLabel : public QLabel...
```

- QCaLabelPlugin class.
  This is based on QCaLabel. It is a light wrapper that adds properties accessible to Qt's Designer using the Q_PROPERTY macro.
  QCaLabelPlugin.h:

```
class QCaLabelPlugin : public QCaLabel {
   ...
    Q_PROPERTY(bool variableAsToolTip READ getVariableAsToolTip
                                       WRITE setVariableAsToolTip)
    Q_PROPERTY(bool enabled READ isEnabled WRITE setEnabled)
    Q_PROPERTY(bool allowDrop READ getAllowDrop WRITE setAllowDrop)
```

- QCaLabelPluginManager class.
  This is based on QDesignerCustomWidgetInterface and provides the factory that will generate QCaLabelPlugin instances as required.
  QCaLabelPluginManager.h:

```
class QCAPLUGINLIBRARYSHARED_EXPORT QCaLabelPluginManager : public
QObject, public QDesignerCustomWidgetInterface {
```

- QCaWidgets class.
  This is based on QDesignerCustomWidgetCollectionInterface and provides the plugin library with a definition of what widgets the plugin library supports.
  QCaDesignerPlugin.cpp:

```
class QCaWidgets: public QObject,
```

```
                    public QDesignerCustomWidgetCollectionInterface {
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetCollectionInterface)
    ...
    virtual QList<QDesignerCustomWidgetInterface*>
                                            customWidgets() const;
    ...
    QList<QDesignerCustomWidgetInterface*> widgets;
};
```

- An instance of the Q_EXPORT_PLUGIN2 macro.
  Among other things, this generates two functions required for every plugin library:
  qt_plugin_query_verification_data ()
  and
  qt_plugin_instance()
  qplugin.h:

```
#  define Q_EXPORT_PLUGIN2(PLUGIN, PLUGINCLASS)      \
       Q_PLUGIN_VERIFICATION_SECTION Q_PLUGIN_VERIFICATION_DATA \
       Q_EXTERN_C Q_DECL_EXPORT \
       const char * Q_STANDARD_CALL \
                           qt_plugin_query_verification_data() \
       { return qt_plugin_verification_data; } \
       Q_EXTERN_C Q_DECL_EXPORT QT_PREPEND_NAMESPACE(QObject) *
Q_STANDARD_CALL qt_plugin_instance() \
       Q_PLUGIN_INSTANCE(PLUGINCLASS)
```

Finding plugins

"Qt applications automatically know which plugins are available, because plugins are stored in the standard plugin subdirectories. Because of this applications don't require any code to find and load plugins, since Qt handles them automatically." C++ GUI Programming with Qt 4.

That is fine, except there can be many 'standard' plugin directories. For example, the QCaplugin library is used by Designer when developing GUIs, the ASgui application to display the GUIs, and by QtCreator when developing the widgets in the first place. Each looks for plugins in a different 'standard' place as follows:

- When Designer is started as a plugin in QtCreator, the QCaPlugin library is loaded from /home/andrew/QtSDK/QtCreator/lib/qtcreator/plugins/designer/
- When Designer is loaded standalone, QCaPlugin library is loaded from /home/andrew/epicsqt/applications/ASguiApp/designer/
- When the QCa GUI application ASgui is run, the QCaPlugin library is loaded from:
  /home/andrew/QtSDK/Desktop/Qt/4.8.0/gcc/plugins/designer/
  or
  /home/andrew/epicsqt/applications/ASguiApp/designer/

Note, an application can add locations where plugins are searched using the static function QCoreApplication::addLibraryPath();

QMake

Qt generates conventional platform specific make files from generic Qt project files.

In the Qt world, the project (.pro) file is considered a source file, and the make files are considered intermediate files.

Plus:

- I have never seen QMake fail to generate a consistent makefile, and I have never been limited by the project file syntax from specifying how I wanted to build a project.
- Platform independence works very well. For example paths names and library specifications are always unix style.
- QMake is run to generate make files every time the project definition changes. After that building is completed with good old fashioned make all the way down.

Minus:

- The project file imposes one level of indirection between you and the make file. This is a problem until you become familiar with the .pro file syntax.
- The generated make files do not seem to handle meta object code well. It is possible to make changes that do not result in regeneration and recompilation of met object code when required.

The following is a cut down project file from the QCa framework - Plugins.pro:

```
QMAKE_CXXFLAGS_DEBUG += -pg
QMAKE_LFLAGS_DEBUG += -pg
QT += core gui multimedia
TEMPLATE = lib
CONFIG +=           plugin \
uitools \
designer \
debug_and_release
DEFINES += QCAPLUGIN_LIBRARY
TARGET = QCaPlugin
OTHER_FILES +=src/QCaSpinBox.png \
src/QCaSlider.png \
src/QCaShape.png \
...
...
HEADERS +=          include/QCaPluginLibrary_global.h \
include/QCaSpinBoxPluginManager.h \
include/QCaSpinBoxPlugin.h \
include/QCaComboBoxPluginManager.h \
...
...

SOURCES +=          src/QCaSpinBoxPluginManager.cpp \
src/QCaSpinBoxPlugin.cpp \
src/QCaComboBoxPluginManager.cpp \
```

```
src/QCaComboBoxPlugin.cpp \
src/QCaSliderPluginManager.cpp \
...
...
RESOURCES +=  src/QCaResources.qrc
INCLUDEPATH +=$$(QCAFRAMEWORK)/plugins/include \
$$(QCAFRAMEWORK)/api/include \
$$(QCAFRAMEWORK)/data/include \
$$(QCAFRAMEWORK)/widgets/include \
$$(QCAFRAMEWORK)/qwt/src \
$$(EPICS_BASE)/include
unix:INCLUDEPATH += $$(EPICS_BASE)/include/os/Linux
win32:INCLUDEPATH += $$(EPICS_BASE)/include/os/WIN32
INCLUDEPATH += $$(EPICS_BASE)/include

LIBS += -L$$(EPICS_BASE)/lib/$$(EPICS_HOST_ARCH) \
-lca \
-lCom \
-L$$(QCAFRAMEWORK)/qwt/lib \
-lqwt

FORMS +=      /widgets/src/PeriodicDialog.ui \
src/PeriodicSetupDialog.ui \
src/PeriodicElementSetupForm.ui
```

Meta object system

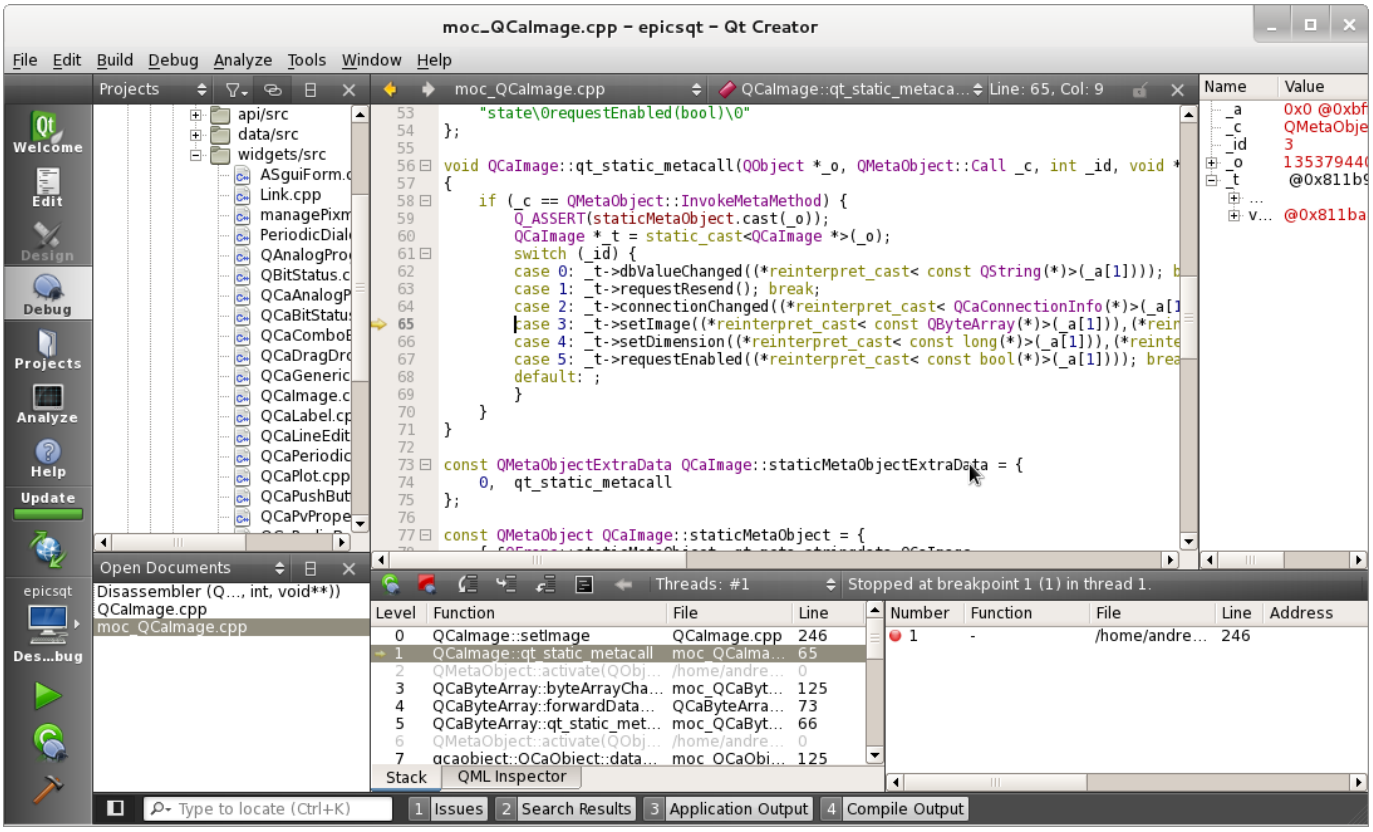
The Qt Meta Object system is how Qt does a lot of its clever stuff.

Signals and slots. QObject. Plugin properties. All rely on the Meta Object Compiler to generate the appropriate C++ code. Similar to Similar to the way dispatch mechanisms are built for COM objects in MFC.

Your code ends up including a lot of moc code you didn't write. When structures change – the make system does not always pick up what moc code needs to be rebuilt resulting in very weird behaviour. If strange stuff starts happening after altering class definitions, then rebuild all.

Moc code generation and compilation can take as much time as building the rest of your code.

It's all there to see in the IDE as follows: Not sure if this is good or bad.



Qt's examples

The SDK comes with a project containing a comprehensive set of 260 (mostly) working examples which is well worth building and exploring. The example demonstrate every class in code designed to explain the class.

A real, working, simple example on your desk top beats some dodgy fragment of code from some internet forum any day.

To build Qt 4.7 examples:

- First apply the following dodgy fix found on an internet forum so all the examples build OK:

  Edit:

  /home/andrew/QtSDK/Examples/4.7/tools/plugandpaint/plugandpaint.pro

  Change:

  ```
  LIBS = -L$${QT_BUILD_TREE}/examples/tools/plugandpaint/plugins -
  lpnp_basictools
  ```

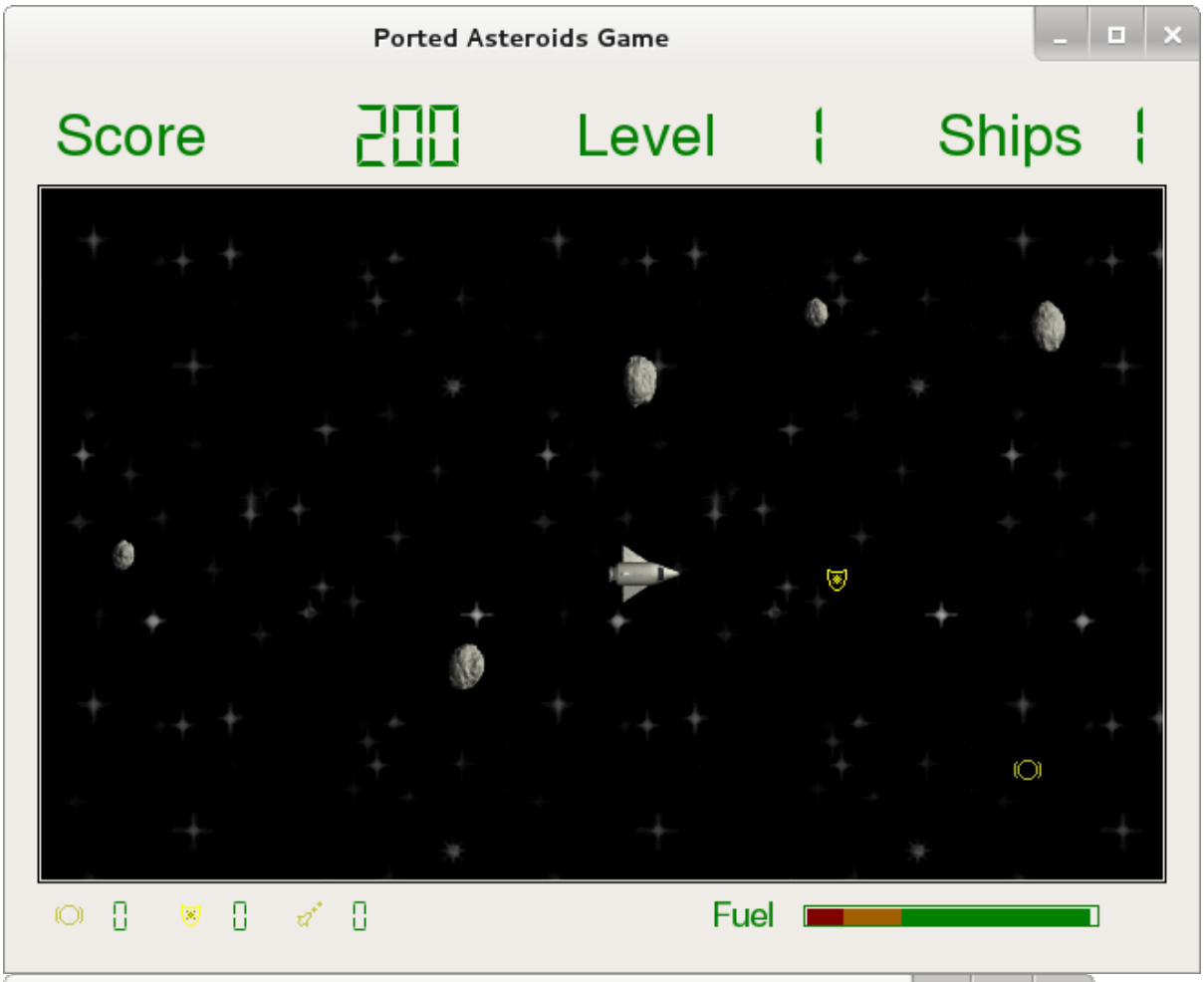
  To:

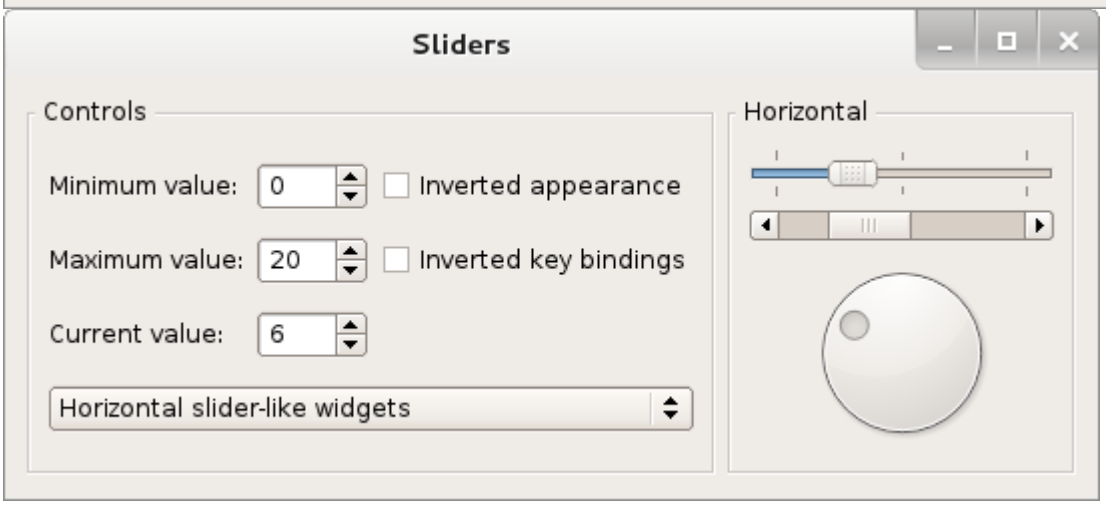
  ```
  LIBS = -L$${QT_BUILD_TREE}/examples/tools/plugandpaintplugins/basicto
  L plugins/ -lpnp_basictools
  ```

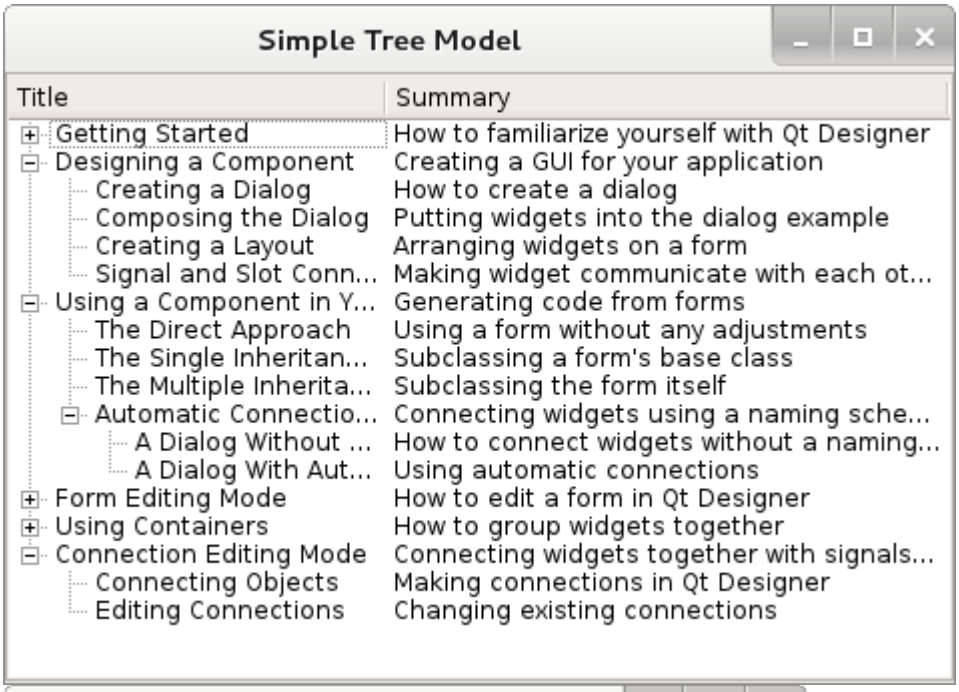
- In Creator, open /home/andrew/QtSDK/Examples/4.7/examples.pro
- Build it

- Select the example to run from the project settings – Run Settings -> Run configuration menu. For example: Mandelbrot
- Run

Some of Qt's examples:







Qt based EPICS drivers, gensub functions

Any IOC code that would benefit from a rich C++ framework can use the Qt library and can be developed within the Qt IDE.

Embed the functionality within a C++ class, then implement that class from within the standard C framework, or base that class on a standard C++ class such as asynPortDriver.

Asub / Gensub

A standard asub or gensub C function can use a C++ class written in Qt.

In the following example, a standard asub function calls a static member of a class written in Qt:

(The class is defined in autogap.cpp and autogap.h in the same directory)

/home/andrew/Insertion_Device_Control/SXR_Insertion_Device_Control_AutoGapSup/src/autogapSub.c:

```
...
#include <aSubRecord.h>
#include <epicsExport.h>
...
...
static long autogapProcess (aSubRecord* prec)
{
    /* Inputs */
    double*        energy   = (double*)        (prec->a);
    unsigned long* harmonic = (unsigned long*) (prec->b);
    double*        angle    = (double*)        (prec->c);
    unsigned long* mode     = (unsigned long*) (prec->d);
    char*          file     = (char*)          (prec->e);
    unsigned long* enabled  = (unsigned long*) (prec->f);

    /* Outputs */
    double* gap    = (double*)(prec->vala);
    double* phase  = (double*)(prec->valb);
    char*   string = (char*)  (prec->valc);

...

    /* calculate gap and phase */
    int status = autogapCalc( *energy, *harmonic, *angle, *mode,
                              file, gap, phase, string, 120 );

    /* Return zero to force transfer of output values to output
       links, or 1 to indicate no change */
    return( status );
}

/* ---------------------------------------------------------- */
epicsRegisterFunction (autogapInit);
epicsRegisterFunction (autogapProcess);
```

Using existing C templates

A standard asyn C template can use a C++ class written in Qt.

The Australian Synchrotron has used a standard C template for creating asyn driver code.

An example is included where the required functionality has been implemented in a Qt C++ class, and the C template has been used as a wrapper around this class.

The code can be found in: /home/andrew/IEM/injEffMonSup/src/

And the C template code is in drvInjEffMon.cpp

Using existing C++ driver templates

"asynPortDriver is a base C++ class that is designed to greatly simplify the task of writing an asyn port driver. It handles all of the details of registering the port driver, registering the supported interfaces, and registering the required interrupt sources"

```
asynPortDriver::asynPortDriver(     const char * portName,
    int  maxAddr,
    int  paramTableSize,
    int  interfaceMask,
    int  interruptMask,
    int  asynFlags,
    int  autoConnect,
    int  priority,
    int  stackSize
)
```

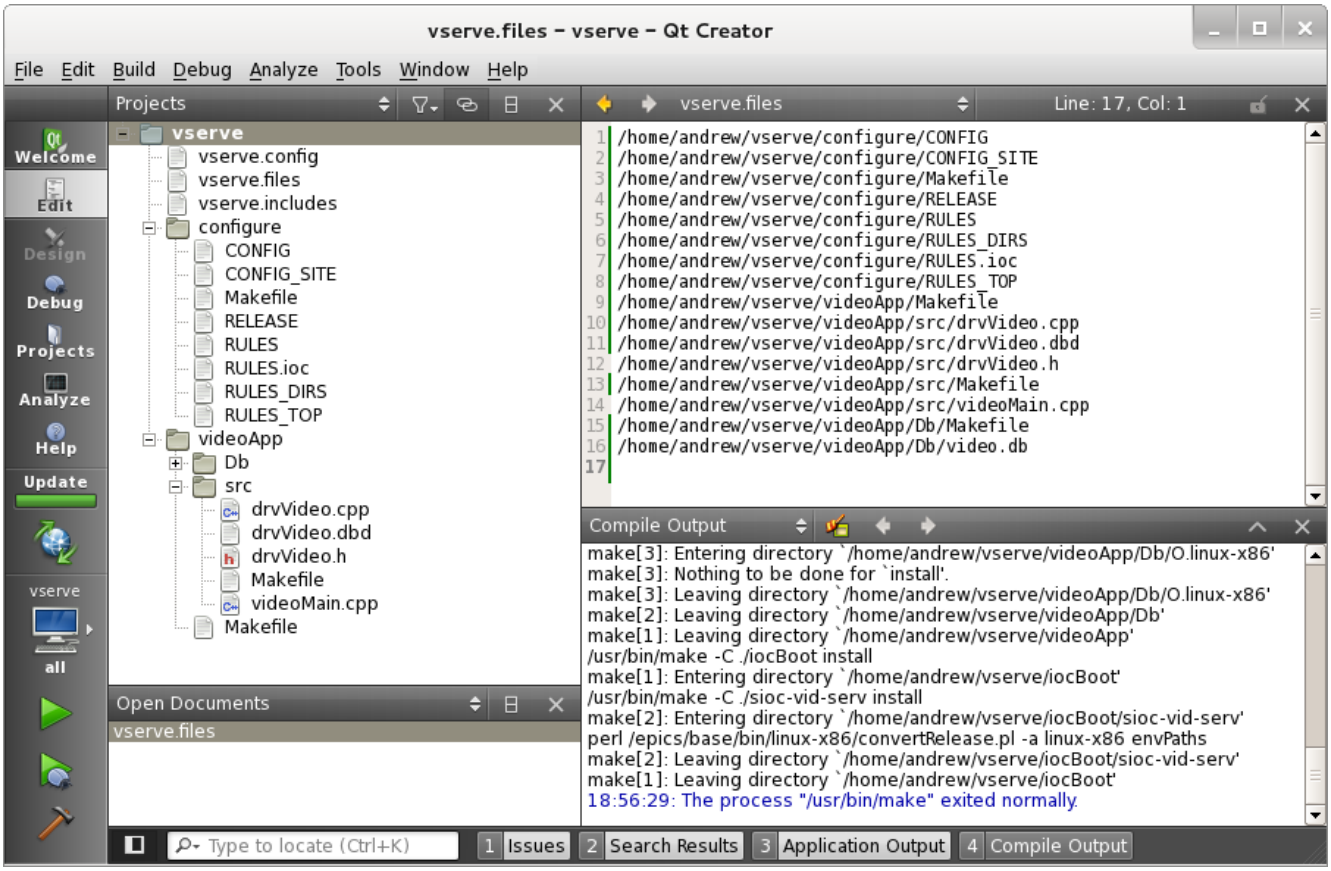
Using Qt's IDE to work with existing, non Qt EPICS code

- Where did it crash?
- What set that variable to that?
- Where did that get called from?
- Where is DBR_TIME_FLOAT defined again?
- Throw another printf in. Shame I had to stop and rebuild to do it.

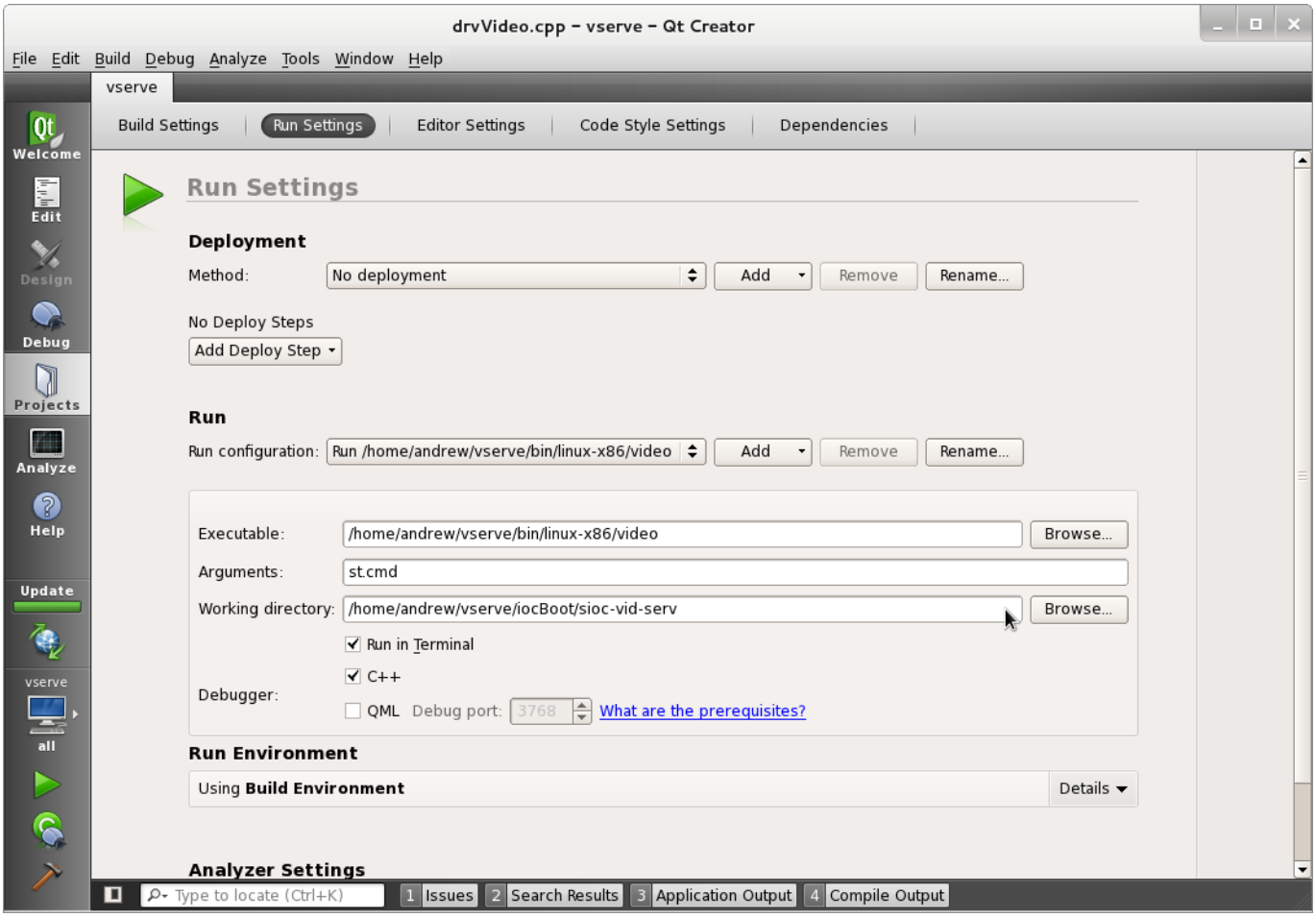
Creator is a great IDE to drop over an exising, non Qt, application.

An example: an IOC supplied by Zen Szalata from SLAC. The following steps were required to be stepping through it in debug within QtCreator:

- From creator, select New File or Project... -> Other Project -> Import Existing Project
- Select top level IOC directory
- Name the project 'vserve'
- Open vserve.files and add what creator hasn't found (such as configure and db, dbd, etc) as follows:



- Set project settings to run an IOC as follows:

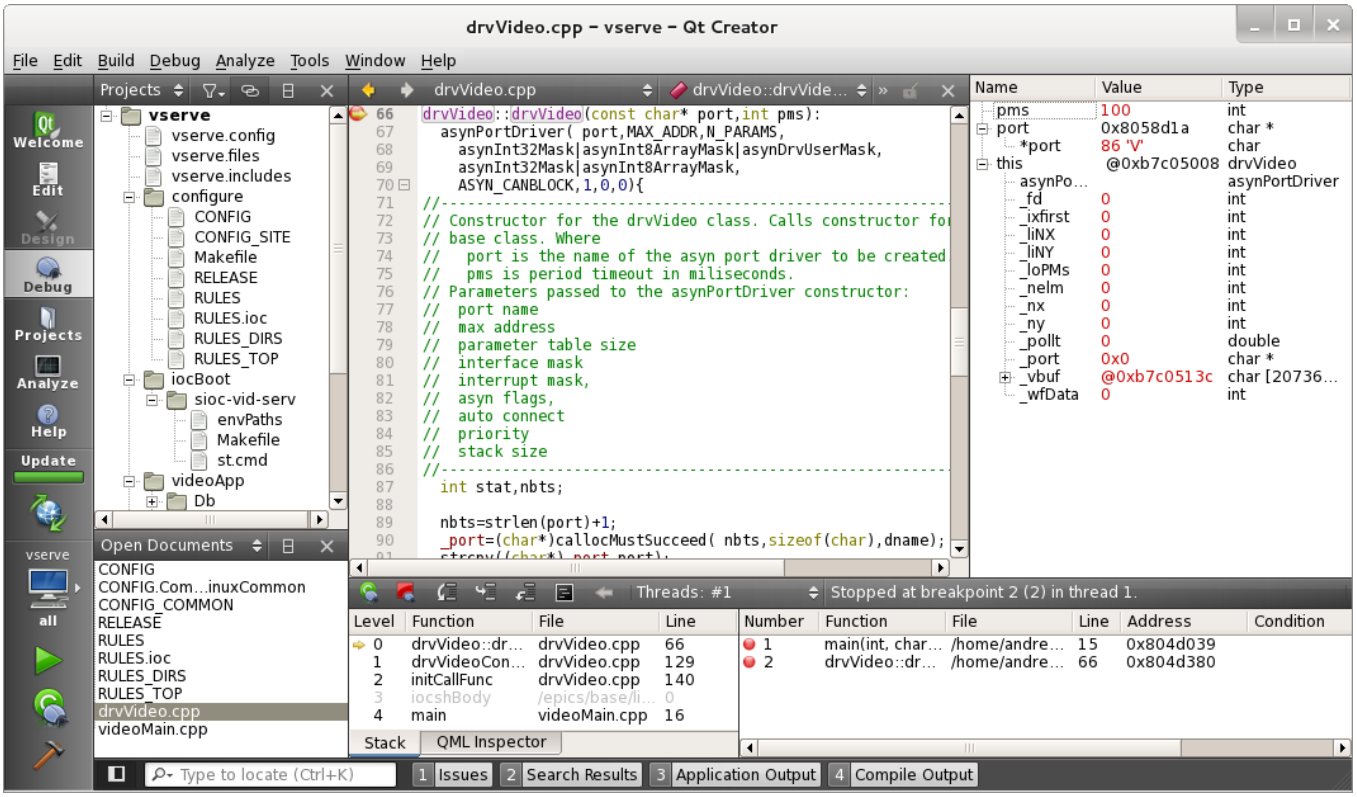


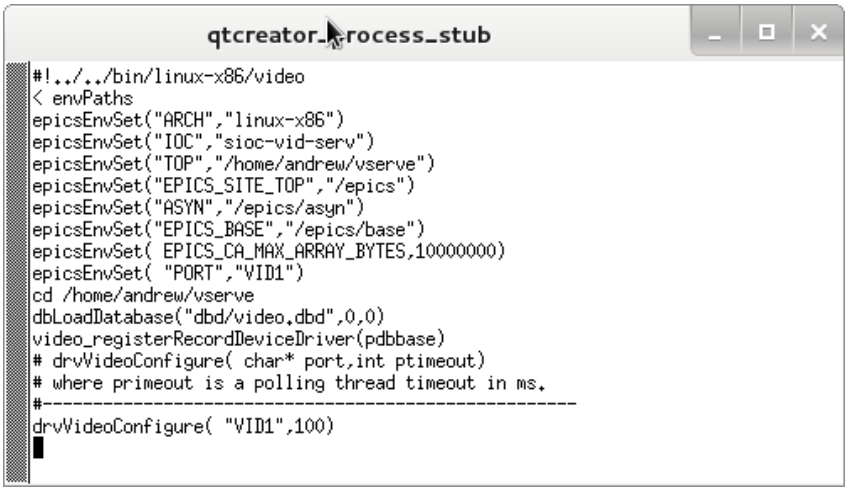
Note, Creator will expect xterm to be installed if asking for the application to run in a terminal.

- If debugging is required, add gcc debug –g flag in In CONFIG.Common.linuxCommon as follows:

```
CODE_CPPFLAGS = -D_REENTRANT -g
```

- Build.
- Start debugging! Below shows the IOC sitting at a break point part way through startup:





QCa overview

QCa design aims

A framework that provides consistent Qt friendly access to CA at the following levels:

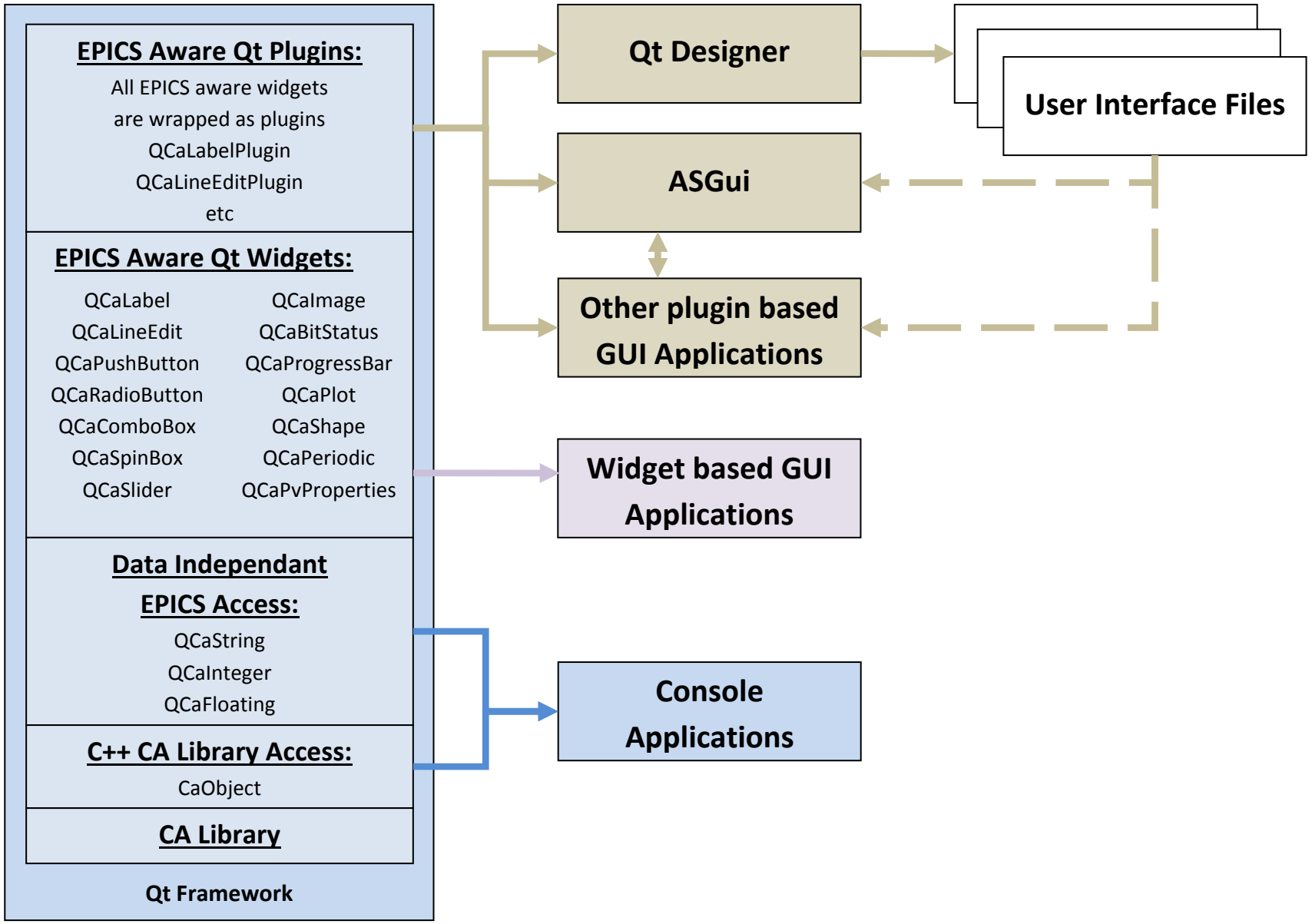
- Code free GUI development using only Qts form designer
- Code rich GUI development using the same widget set, but also offering the same Qt based CA data access used by the widgets.
- Non-GUI console application and programs, again using the same Qt based CA data access.

The QCaFramework is designed to hide unwanted complexities of CA. At its simplest a PV is supplied and data update signals are delivered. Or not. More details are available when requested, but without the need to

delve into any CA structures. For example, alarm and status signals are available, and methods present to return EGU, limits, etc. Data can be provided with different levels of interpretation:

- Raw data packaged in a QByteArray
- Data packaged as a QVariant. Depending on the underlying CA record, the QVariant will contain an integer, a floating point number, or a QString.
- Data packaged always as an intger, floating, or QString regardless of the underlying CA data type. Classes are available to direct how conversion takes place.

QCa framework class hierarchy



Type of access to CA data	Functionality	Main classes	
C++ access to the CA library.	Provides convenient C++ access to the CA library.	CaObject	
Qt based access to CA.	Hides CA specific functionality. Adds Qt functionality such as signals and slots.	QCaObject	
Data type independent access.	Hides EPICS data types, providing read and write conversions where required.	QCaInteger QCaString	QCaFloating QCaByteArray
EPICS aware graphical widgets.	Implements graphical Qt based widgets that provide access to EPICS data.	QCaLabel QCaLineEdit QCaPushButton QCaRadioButton QCaComboBox QCaSpinBox QCaSlider	QCaImage QCaBitStatus QCaProgressBar QCaPlot QCaShape QCaPeriodic QCaPvProperties
EPICS aware graphical Qt plugins.	Adds Qt plugin interfaces to EPICS aware widgets.	QCaLabelPlugin QCaLineEditPlugin QCaPushButtonPlugin QCaRadioButtonPlugin QCaComboBoxPlugin QCaSpinBoxPlugin QCaSliderPlugin	QCaImagePlugin QCaBitStatusPlugin QCaProgressBarPlugin QCaPlotPlugin QCaShapePlugin QCaPeriodicPlugin QCaPvPropertiesPlugin
GUI support widgets	Implements Qt based widgets that support control system GUIs. These widgets to not access the CA library.	AsGuiForm QCaPushButton	Link QCaSubstitutedLabel

Wrapping CA in a C++ class

CaObject

The CaObject class provides a C++ interface to CA.

It is a very light wrapper and does add any functionality except provide a C++ interface to CA.

As it did not need to add any rich functionality that could benefit from Qt's classes it was written in pure C++. The idea was to make it useable to a wider audience that Qt developers, but I think it is too light on its own to be of much use. Within the QCa framework it is used as a base class for QCaObject.

It has three sets of states that a child class like QCaObject must keep track of:

- Channel state held in the CaConnection class
- Link state held in the CaConnection class
- Process state held in the CaRecord class

CaConnection.h:

```
namespace caconnection {

  enum link_states { LINK_UP, LINK_DOWN, LINK_UNKNOWN };
  enum channel_states { NEVER_CONNECTED, PREVIOUSLY_CONNECTED,
CONNECTED, CLOSED, CHANNEL_UNKNOWN };
```

CaRecord.h

```
namespace carecord {

  enum process_state { NO_UPDATE, FIRST_UPDATE, UPDATE };
```

QCaObject

The QCaObject class is based on the caObject class.

It manages the CA connection state, link state, and all read and write processing.

Users of the QCaObject class access CA data in a Qt friendly way: All CA callbacks are translated to Qt signals, with the data encapsulated in a QtVariant class

A caller only needs to provide a QCaObject class with a PV name connect to its data and status signals, then request that a subscription be established, or that a single read completed.

Classes based on QCaObject do not have to include any EPICS definitions (cadef.h, etc)

There are several classes within the QCa framework that are based on QCaObject. The following is a paraphrase of code from QCaInteger.

```
QCaInteger.cpp
```

```
// Construction

QCaInteger::QCaInteger( QString recordName, QObject *eventObject,
                        QCaIntegerFormatting *integerFormattingIn,
                        unsigned int variableIndexIn ) : QCaObject(
recordName, eventObject ) {

...
...

    QObject::connect(
           this,
           SIGNAL( connectionChanged(  QCaConnectionInfo& ) ),
           this,
           SLOT( forwardConnectionChanged( QCaConnectionInfo& ) ) );

    QObject::connect(
           this,
           SIGNAL( dataChanged( const QVariant&, QCaAlarmInfo&,
                                QCaDateTime& ) ),
           this,
           SLOT( convertVariant( const QVariant&, QCaAlarmInfo&,
                                 QCaDateTime& ) ) );
}

// Writing data
void QCaInteger::writeInteger( const long &data ) {
    writeData( integerFormat->formatValue( data, getDataType() ) );
}

// Data update
void QCaInteger::convertVariant( const QVariant &value, QCaAlarmInfo&
alarmInfo, QCaDateTime& timeStamp ) {
    emit integerChanged( integerFormat->formatInteger( value ),
                         alarmInfo, timeStamp, variableIndex );
}

// Status update
void QCaInteger::forwardConnectionChanged( QCaConnectionInfo&
connectionInfo) {
    emit integerConnectionChanged( connectionInfo, variableIndex );
}
```

When a CA connection is attempted, it should complete eventually, even if the CA server is down, the connection should complete when the server finally starts.

The QCaObject does not wait forever. It re-attempts the connection every 60 seconds. This handles the case where a gateway may lose the request as reportedly can happen if the gateway itself is restarted while the CA server is down.

The user of the QCAObject class does not need to be aware of CA data types. All data is supplied as a QVariant. A QVariant can be one of around 50 types. QCaObject only needs the following Qt types and pure C++ to package all CA data:

- qlonglong
- qulonglong
- double
- QString
- QVariantList (containing homogeneous list of one of the above types)

The QCaObject class provides a set of methods to obtain context information about the PV, such as alarm limits, as follows:

```
QString getEgu();
QStringList getEnumerations();
unsigned int getPrecision();
double getDisplayLimitUpper();
double getDisplayLimitLower();
double getAlarmLimitUpper();
double getAlarmLimitLower();
double getWarningLimitUpper();
double getWarningLimitLower();
double getControlLimitUpper();
double getControlLimitLower();
```

CA writes can be completed with or without callbacks. The QCaObject class can perform either type. If callbacks are required the QCaObject class will translate write completion callbacks to a Qt signal. Write callbacks are enabled using the following function:

```
void QCaObject::enableWriteCallbacks( bool enable )
```

This is not just for better write status, if affects the behaviour of the write. When using write with callback the record will finish processing before accepting the next write. Writing with callback may be required when writing code that is tightly integrated with record processing and the code needs to know processing has completed.

Writing with no callback is more desirable when a detachment from record processing is required, for example in a GUI after issuing a motor record move a motor stop command will take effect immediately if writing without callback, but will (erroneously) only take effect after the move has finished if writing with callback.

An abridged version of the QCaObject class definition is as follows:

```
QCaObject.h

  class QCAPLUGINLIBRARYSHARED_EXPORT QCaObject :
                                      public QObject,
                                      caobject::CaObject {

    public:
      QCaObject( const QString& recordName, QObject *eventObject,
                 unsigned char signalsToSendIn=SIG_VARIANT );
      QCaObject( const QString& recordName, QObject *eventObject,
                 UserMessage* userMessageIn,
                 unsigned char signalsToSendIn=SIG_VARIANT );

      bool subscribe();
      bool singleShotRead();

      void enableWriteCallbacks( bool enable );
      bool isWriteCallbacksEnabled();

      // Get database information relating to the variable
      QString getEgu();
      QStringList getEnumerations();
      unsigned int getPrecision();
      double getDisplayLimitUpper();
      double getDisplayLimitLower();
      double getAlarmLimitUpper();
      double getAlarmLimitLower();
      double getWarningLimitUpper();
      double getWarningLimitLower();
      double getControlLimitUpper();
      double getControlLimitLower();

    signals:
      void dataChanged( const QVariant& value, QCaAlarmInfo&
                        alarmInfo, QCaDateTime& timeStamp );
      void dataChanged( const QByteArray& value, QCaAlarmInfo&
                        alarmInfo, QCaDateTime& timeStamp );
      void connectionChanged( QCaConnectionInfo& connectionInfo );

    public slots:
      bool writeData( const QVariant& value );
      void resendLastData();
```

qepicspv class: Non-event driven access to CA data.

The QCaObject class is event driven; it will generate signals when data arrives. For some applications this is not the best model. The qepicsph class is a light wrapper around the QCaObject class. It provides simple blocking or non blocking read and write functions.

The user provides a PV and proceeds to read or write to the class at their leisure. The application does not need to respond to signals.

The qepicspv class provides even simpler usage with static read and write functions. Typical usage is as follows:

```
qepicspv::set( "MY_PV", 123456);
int value = qepicspv::get( "MY_PV" );
```

An abridged version of the qepicspv class definition is as follows:

```
qepicspv.h

class QCAPLUGINLIBRARYSHARED_EXPORT QEpicsPV : public QObject {

public:

  QEpicsPV(const QString & _pvName, QObject *parent = 0);

  const QVariant & get() const;
  const QVariant & getUpdated(int delay=defaultDelay) const;

  bool isConnected() const;

  static QVariant get(const QString & _pvName, int
                      delay=defaultDelay);
  static QVariant set(QString & _pvName, const QVariant & value,
                      int delay = -1);
}
```

Translating CA callbacks to Qt signals

CA callbacks occur in the context of an EPICS thread. It is generally unsafe to call any Qt class method except from a Qt thread. About the only safe task is posting an event to a Qt event queue. The event can then be picked up by a Qt event handler and delivered in the context of a Qt thread, at which point a class can use the Qt framework to emit a signal.

The process of translating a CA callback to a signal is not trivial, but is reasonably efficient. One exception is large array data where the data must be copied. Prior to returning from the CA callback.

In the QCa framework, the translation from EPICS thread callback to Qt signal described above is implemented in the following steps: (Note, code fragments are heavily edited)

- Process the callback within the context of the EPICS thread:
    - Catch the callback in a static class method of CaObject, generate a reference to the originating class instance from the callback arguments, and call a method the originating QCaObject class

instance.

CaObject.cpp:

```
(static)
void CaObjectPrivate::readHandler( struct event_handler_args args ) {
   ...
   CaObject* context = (CaObject*)args.usr;
   CaObjectPrivate* p = (CaObjectPrivate*)(context->priPtr);
   ...
   p->processChannel( args );
   context->signalCallback( READ_SUCCESS );
```

- Build a QEvent and post it on to Qt event queue

  QCaObject.cpp:

```
void QCaObject::signalCallback( caobject::callback_reasons newReason )
{
   dataPackage = getRecordCopyPtr();
   QCaEventUpdate* newDataEvent = new QCaEventUpdate( this,
                                                      newReason, dataPackage );
        QCaEventItem item( newDataEvent );
        QCoreApplication::postEvent( eventHandler, newDataEvent );
```

- Process the event within the context of the Qt event loop

  - Catch the event in an event filter added to the Qt event system and pass the event on to a static method of QCaObject

    QCaEventFilter.cpp:

```
(static)
bool QCaEventFilter::eventFilter( QObject *watched, QEvent *e ) {
  if( e->type() == QCaEventUpdate::EVENT_UPDATE_TYPE ) {
   QCaEventUpdate *dataUpdateEvent = static_cast<QCaEventUpdate*>( e );
   qcaobject::QCaObject::processEventStatic( dataUpdateEvent );
   ...
```

  - Extract the reference to the originating QCaObject class instance and call a method in the QCaObject class instance that posted the event

    QCaObject.cpp:

```
(static)
void QCaObject::processEventStatic( QCaEventUpdate* dataUpdateEvent )
{
   ...
   dataUpdateEvent->emitterObject->processEvent( dataUpdateEvent );
   ...
```

  - Emit a Qt signal from the QCaObject class instance.

    QCaObject.cpp:

```
void QCaObject::processEvent( QCaEventUpdate* dataUpdateEvent ) {
   ...
   emit dataChanged( value, alarmInfo, timeStamp );
   ...
```

Managing data overloads

If CA callbacks arrive quicker than they can be processed, the backlog occurs in the form of a growing Qt event queue.

I have not explored what limits are imposed by Qt on its event queues, but if Qt does not manage it, memory limits will be reached eventually, and long before then the data being processed from the event queue may be stale, depending on the use. For example it may be OK for an archiving function to get behind as long as there are quiet periods where it can catch up, but a GUI can't afford to have data changing in front of a user that is not current.

The QCaObject managed this backlog by keeping track of unprocessed events in the queue. If there is an unprocessed event in the queue when more data arrived, the data payload in the unprocessed event is updated with the newer data.

This has the following benefits:

- No matter how busy the system, stale events do not occur.
- Since the task of updating an event's payload is quick, generally, compared to processing the event a very large overload can be accommodated.

However:

- Updates can be silently discarded. For some applications this may be unacceptable.

This system was only designed with GUIs in mind. The QCaObject class should probably be enhanced to allow specifying the number of unprocessed events per QCaObject. It should also be possible to recognise data is being discarded.

The code that manages data overloads is in QCaObject.cpp:

```
void QCaObject::signalCallback( caobject::callback_reasons newReason )
```

QCa design issues and solutions

Rogue CA callbacks.

An outstanding problem with the QCaFramwork is CA callbacks that occur after a channel has been closed and the supporting QCa classes destroyed. Apart from confirming this actually occurs, I haven't investigated this much. No idea if I'm closing down the CA channel properly or not.

CaConnection .cpp:

```
void CaConnection::shutdown() {
    CA_UNIQUE_CONNECTION_ID--;

    if( channel.activated == true ) {
        ca_clear_channel( channel.id );
    }
    if( context.activated == true ) {
        if( CA_UNIQUE_CONNECTION_ID <= 0 ) {
            ca_context_destroy();
```

```
            }
        }
}
```

Providing access to raw data.

The QCaObject class usually supplies data signals with the data packaged in a Qt friendly way - always a QVariant containing either interger, floating or string data, or an QVariantList of the same.

The QCaObject class can also supply the raw data packaged in a QByteArray.

Returning raw data was added to support the QCaImage widget. This is an example where the raw data is best processed by the end consumer. Early conversion to a simple QVariantList was time consuming and an expensive halfway step towards generating a QImage from the data.

An alternative would have been to perform the conversion to a QImage within the QCaObject class. The main arguments against this are:

- When packaging the data in the QCaObject class, the only information about the data comes from the CA record which only says it is an array. It is not explicitly an image.
- Even when the data is an image, the processing required by the end consumer may not be simply image presentation and generation of a QImage may itself be an unnecessary overhead.

Application interaction with plugin widgets

QCa framework widgets are designed to operate independent of their environment. Give them a PV and off they go. It doesn't matter if they are being manipulated in Qts Designer, present in a user interface file opened by the ASgui application, or created in any other program. How then do they interact effectively with the application they find themselves in. For example,

- How does an EPICS aware push button widget request the application to open a different GUI?
- How does an EPICS aware widget log error messages in the application status window, assuming there is one?
- How does the application supply EPICS macro substitutions to widgets it is not even aware it has created? Remember, ASgui uses QLoader to load .ui files. It doesn't know if there are any QCa widgets present, and if there are they may be buried in nested sub forms.

The QCa framework manages this using the ContainerProfile class. (The class name means a profile of whatever is containing the QCa widgets.)

Notes from ContainerProfile.cpp:

This class provides a communication mechanism from the code creating QCa widgets to the QCa widgets. When QCa widgets, such as QCaLabel, are created, they need to know environmental information such as what macro substitutions to apply, or where to signal error messages.
Also, the code creating the QCa widgets may require a reference to all the created QCa widgets.
In some cases this information cannot be passed during construction or set up post construction via a method. For example, when the object is being created from a UI file by Qt. In this case the application code asks Qt to generate objects from a UI file and has no idea what QCa widgets if any have been created.

To use this class, an instance of this class is instantiated prior to creating the QCa widgets.
Information to be communicated such as message handlers and macro substitutions is set up within this class. Then the QCa widgets are created using a mechanism such as the QUiLoader class.
As each QCa widget is created it also instantiates an instance of the ContainerProfile class. If any information has been provided, it can then be used.
Note, a local copy of the environment profile is saved per instance, so an application creating QCa widgets (the container) can define a profile, create QCa widgets, then release the profile.
To use this class

- Instantiate a ContainerProfile class
- Call setupProfile()
- Create QCa widgets
- Call releaseProfile()

Exercises:

Simple

Using Creator create a simple console app to log updates from a PV

Worked example: /home/andrew/simple/simple.pro

Slider

Using Creator create a GUI application with a CA aware slider widget writing to a PV

Worked example: /home/andrew/slider/slider.pro

Notes:

- The slider is also updated when the PV value changes.
- The PV hard coded in the worked example is MY:ai
  One the demo VirtualBox machine there is a button on the top panel to start 'My IOC" which is in /home/Andrew/myIOC

No Code

Using Designer, create a UI file that can be opened by the ASgui display application.

Worked examples:

- /home/Andrew/shape.ui
- /home/Andrew/CaEnabledConventionalWidgets.ui
- /home/Andrew/CaEnabledSpecialistWidgets.ui

Note:

- These examples use PVs in 'My IOC'
  One the demo VirtualBox machine there is a button on the top panel to start 'My IOC" which is in /home/Andrew/myIOC

Perhaps:

- Add standard Qt widgets (including a web browser?)
- Add some signals and slots

QCa framework on SourceForge

The QCa framework is available at http://sourceforge.net/projects/epicsqt/