15th June 1025 Note:

While this historical document provides a useful overview of the EPICSQt framework it is not an active document and there have been considerable changes to the framework since it was written in 2009.

In particular, the framework is now referred to as the EPICSQt framework, or QE framework and the prefix in EPICS aware widget names has changed from QCa to QE.

Also, the framework has been extended considerably since this document was produced with many new widgets and supporting classes.

Please refer to 'QE Framework - QEGui and User Interface Design' (QE_QEGuiAndUserInterfaceDesign.pdf) for an up to date description of the entire widget set.

Australian Synchrotron Technical Note AS-200906-01

Qt CA Framework

Author: Andrew Rhyder

26 June 2009

Abstract

Description of a software framework based on Qt for accessing EPICS data using Channel Access, highlighting access to the various layers of the framework.

1 Introduction

Channel Access is described as 'one of the core components of an EPICS system. It is the software component that that allows a Channel Access client application to access control-system data which may be located on different hosts throughout a network' [1]

While CA is the default means to access EPICS data, its use is not trivial. A significant understanding of how CA works is required to execute the steps required to read or write data. The complexity of setting up and terminating CA requests leaves room for error. Also, CA uses a C programming interface and so does not make use of any object oriented programming techniques.

Qt is a cross-platform application and UI framework. It includes a C++ class library and a cross-platform IDE.

The QCa framework provides a Qt based C++ framework for easy CA access to EPICS data.

It provides access to EPICS data at several levels from programmatic reading and writing of data through to EPICS aware Qt user interface plugins such as push buttons, sliders, and text widgets.

2 QCa Framework Hierarchy Overview

The QCa framework is designed to allow access to CA data in the most appropriate form. The framework is based on a hierarchy of classes as shown in Table 1. This hierarchy is open at all levels to the developer. Appropriate use of the hierarchy is shown in Figure 7.

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
EPICS aware graphical widgets.	Implements graphical Qt based widgets that provide access to EPICS data.	QCaLabel QCaLineEdit QCaPushButton QCaShape QCaSlider QCaSpinBox
EPICS aware graphical Qt plugins.	Adds Qt plugin interfaces to EPICS aware widgets.	QCaLabelPlugin QCaLineEditPlugin QCaPushButtonPlugin QCaShapePlugin QCaSliderPlugin QCaSpinBoxPlugin
GUI support widgets	Implements Qt based widgets that support control system GUIs. These widgets to not access the CA library.	AsGuiForm GuiPushButton CmdPushButton

Table 1QCa framework hierarchy

2.1 C++ access to the CA library - CaObject

The CaObject base class provides a C++ wrapper around the CA library. While available to the developer, it was written mainly to provide a level of abstraction within the Qt based QCaObject class. It is recommended to be used where a Qt framework is not available.

While the CaObject class hides little CA functionality, it does use object oriented techniques to simplify CA access. All CA call-backs are routed through a single pure virtual function. Data is encapsulated in the 'generic' class that provides functions for extracting and interpreting the data stored in the native CA record.

CaObject was produced to support the QCaObject class so QCaObject is a good example of using the CaObject class. The code in Figure 1 shows QCaObject's use of

CaObject as a base class, and implementation of CaObject's call-back pure virtual function `signalCallback()`.

(extract from QCaObject.h)

```
    class QCaObject : public QObject, caobject::CaObject {
            …

            …
          void signalCallback( caobject::callback_reasons reason );
            …
```

Figure 1 Code using CaObject

2.2 Qt based access to CA – QCaObject

The QcaObject class provides full access to EPICS data while hiding most CA specific functionality such as link status, connections and channels.

The QcaObject class adds Qt functionality. Data can be written using a Qt slot and Qt signals are available for data and status information as required. Qt data types to represent all EPICS data.

QCaObject was produced to support data type independent classes such as QCaInteger, so QCaInteger is a good example of using the QCaObject class. The code in Figure 2 shows QCaInteger's use of QCaObject as a base class, and setting up to use data update signals from the QCaObject class. The data in the update signals may be of any type and is represented by a Qt variant.

(extract from QCaInteger.h
Using QCaObject as a base class)

```
   class QCaInteger : public qcaobject::QCaObject {
         …
         …
```

(extract from QCaInteger.cpp
connecting to data update signal from QCaObject)

```
   void QCaInteger::initialise( …
         …
         …
       QObject::connect(this,
                         SIGNAL( dataChanged( const QVariant & ) ),
                         this,
                         SLOT( convertVariant( const QVariant & ) ) );
   }
```

Figure 2 Code using QCaObject

2.3 Data type independent access – QCaInteger, etc

The classes QCaInteger, QCaString, and QCaFloating are based on QCaObject and interpret all data as integers, strings, and floating point numbers respectively. They are used to provide access to EPICS data in a known format regardless of the actual data type of the EPICS data. For example, string data is always required for a text label regardless of the underlying EPICS data type. While some conversions are unlikely to be of much practical use, all conversions are permitted.

Each class is supported by a class defining the conversion and formatting required. For example, a QCaString instance can be set up to always show a leading zero. The conversion requirements also define how to handle conversion errors.

The example code in Figure 3 shows a class which logs updates to the process variable 'QT:A1'. It creates an instance of a QCaString object that generates a signal each time the data changes. The signal contains text regardless of the data type of the process variable. The code directly related to generating and consuming the stream of text updates is highlighted.

```
#include <QObject>
#include <QCaString.h>

class test : public QObject
{
      Q_OBJECT

public:
    test() {
        stream = new QTextStream( stdout );

        source = new QCaString( "QT:A1", this, &formatting, 1,
                                &messages );
        QObject::connect( source,
                          SIGNAL( stringChanged( const QString& ) ),
                          this,
                          SLOT( log( const QString & ) ) );
        source->subscribe();
    }

private:
    QCaString* source;
    UserMessage messages;
    QCaStringFormatting formatting;
    QTextStream* stream;

private slots:
    void log( const QString &data  ){
        *stream << data << "\n";
        stream->flush();
    }
};
```

Figure 3Sample code printing a value whenever a process variable updates

2.4 EPICS aware graphical widgets – QCaLabel, etc

The classes QCaLabel, QCaLineEdit, QCaPushButton, QCaShape, QCaSlider, and QCaSpinBox allow an application to add graphical objects to a user interface that are EPICS aware. That is, they interact directly with EPICS data. The application sets up the EPICS process variable name and other parameters that define how the widget interacts with EPICS data. The application does not have to handle EPICS data or any aspect of the CA interface.

The application may supply the EPICS aware widgets with an object that the widgets can send Qt signals to, including error and status messages signals.

Code in Figure 4 demonstrates the creation of a label displaying data from an EPICS process variable and a slider writing to that process variable. Figure 5 shows the resultant GUI.

```
#include "mainwindow.h"

#include <QCaLabel.h>
#include <QCaSlider.h>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QCaLabel* label = new QCaLabel( "QT:A1", this );
    label->activate();
    QCaSlider* slider = new QCaSlider( /*"QT:A1",*/ this );
    slider->setVariableName( "QT:A1", 0 );
    slider->activate();
    slider->setMinimumHeight( 100 );
}
```

Figure 4Code generating a very simple control GUI

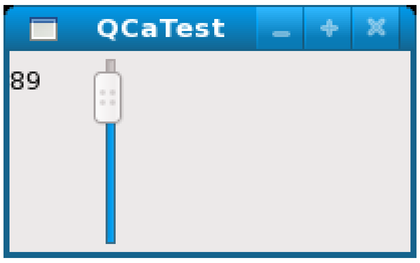

Figure 5Control GUI from code in Figure 4

2.5 EPICS aware graphical Qt plugins – QCaLabelPlugin, etc

The classes QCaLabelPlugin, QCaLineEditPlugin, QCaPushButtonPlugin, QCaShapePlugin, QCaSliderPlugin, and QCaSpinBoxPlugin are EPICS aware widgets with a Qt plugin interface.

These plugins can be used by any Qt application that can load plugins.

They are loaded into the Qt GUI design tool 'designer' which can be used to generate GUI description files that include EPICS aware widgets. These files can be loaded at run time by any application code, or used as source for any application. One application that loads these files at run time in is AsGui.

Examples of the EPICS aware Qt plugins are shown in Figure 6.

2.6 GUI support widgets – AsGuiForm, etc

The classes AsGuiForm, GuiPushButton, and CmdPushButton implement Qt based widgets that support the development of EPICS control system GUIs. They are not EPICS aware widgets.

The AsGuiForm class is based on a Qt scroll area widget (QScrollArea) and can contain any Qt based widgets, including the QCa framework's widgets. The AsGuiForm class provides default mechanisms for using the signals the QCa framework's widgets can generate. It is used as the scroll area in the AsGui application and can be used to create sub forms when developing control system GUIs in 'designer'.

The GuiPushButton class is based on a Qt push button (QPushButton) and is used to launch new GUIs. It can be supplied with a Qt object that will accept Qt signals to start a new GUI or it can use its own default mechanism for creating a new GUI.

The CmdPushButton class is based on a Qt push button (QPushButton) and is used to execute any command. Typically it would be used within a GUI to perform an action on the local machine, such as launch another application.

Examples of the GUI support Qt plugins are shown in Figure 6.

QCa based applications

The QCa framework currently includes a couple of applications. The main application is AsGui.

AsGui is a graphical control system user interface. It displays EPICS aware GUIs based on user interface files created using 'designer' as shown in Figure 6.

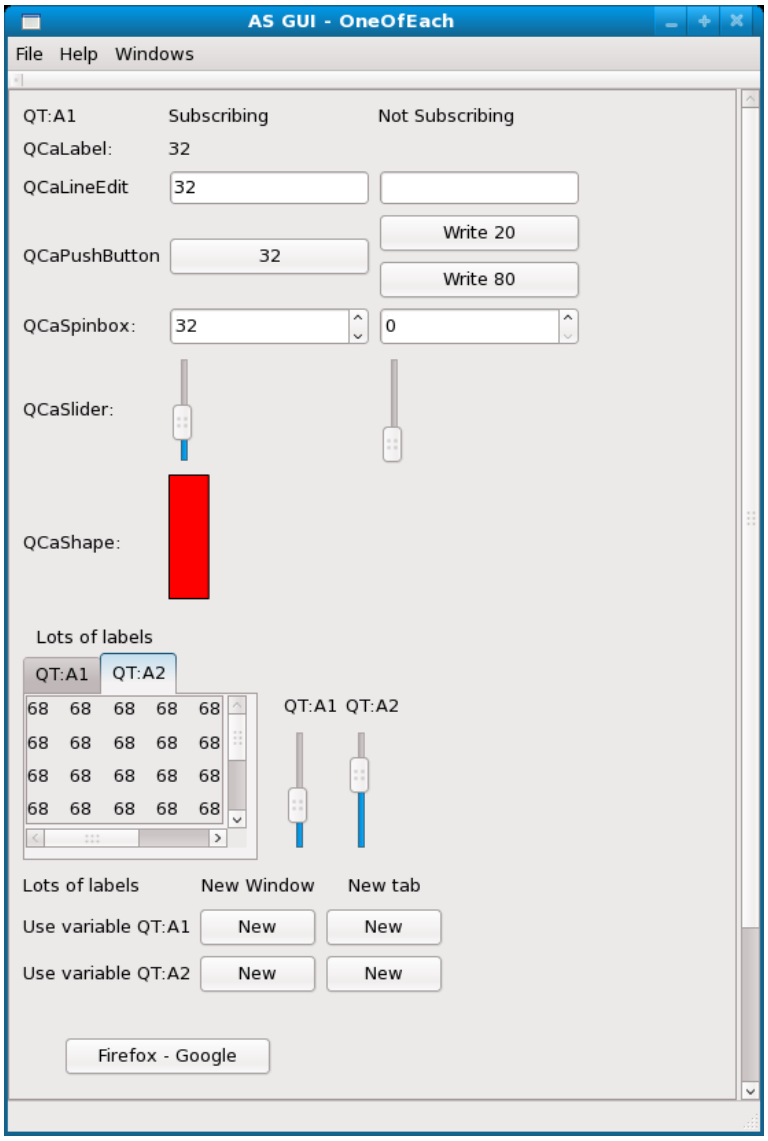

Figure 6A sample GUI created in designer using EPICS aware plugins and GUI support plugins

QCaMonitor is a console application that takes a list of EPICS process variable names as an argument and monitors changes to the data specified by the names. It will perform the same task as the standard EPICS application caget. It is an example of using QCaString objects to generate a stream of textual based updates.

3 Class usage

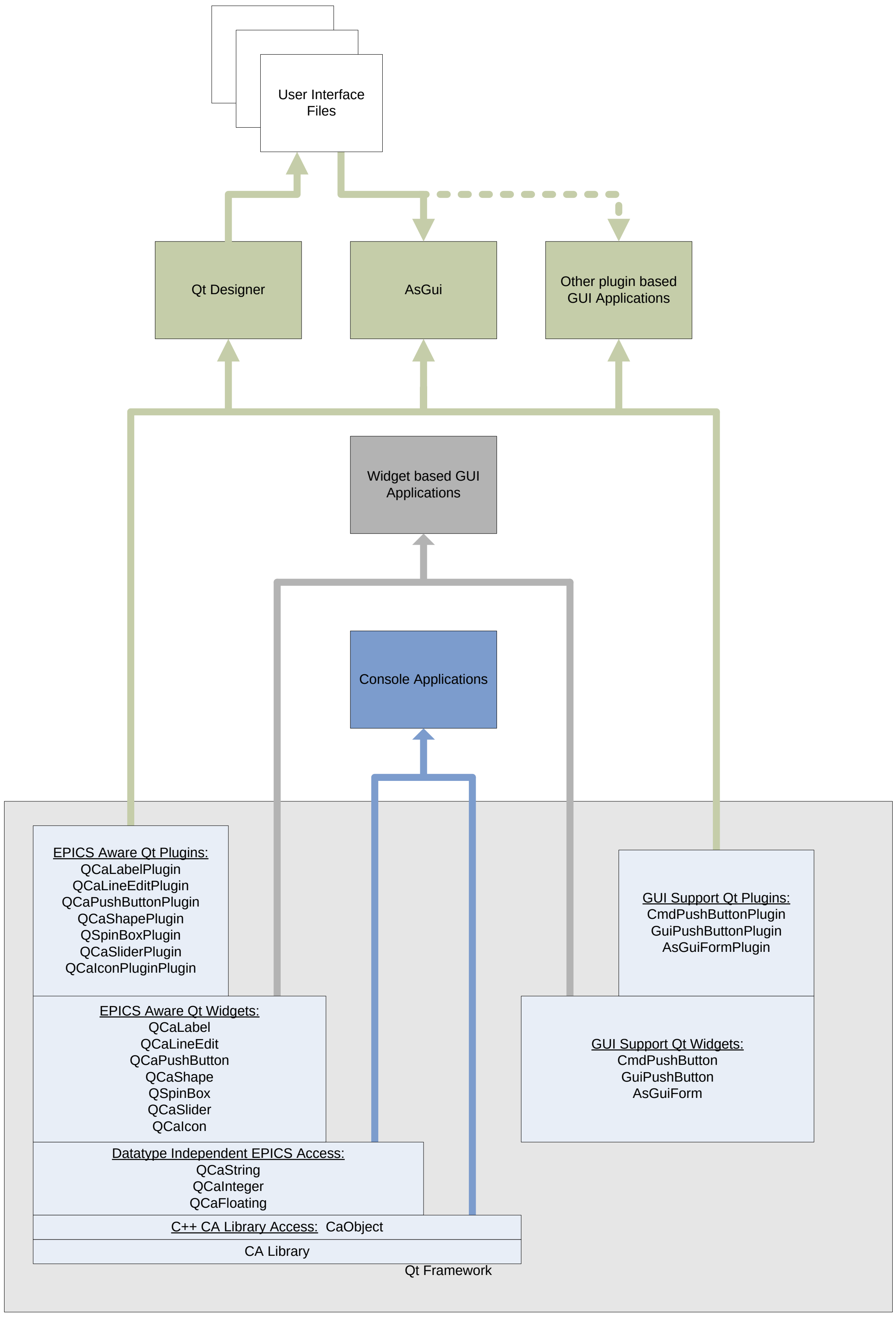

Figure 7Typical QCa class usage

4 QCa Framework Documentation

A Doxygen documentation set is available at:

http://qt/

Project documentation is available at:

K:\Projects\2076 - QT_GUI_Framework\Technical documents

The following project documents are available:

- Requirements Specification
  AS-DOC-0300_Rev0-Qt Requirements Specification.doc
- Design Specification
  AS-DOC-0301_Rev0-Qt Design Specification.doc
- Installation Procedure
  AS-DOC-0302_Rev0-Qt Installation Procedure.doc
- Developers Guide
  AS-DOC-0303_Rev0-Qt Developers Guide.doc

Qt documentation is available at:

http://www.qtsoftware.com/